AD-A202 646

FILE COPY

1

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

89 1 17 179

1

AFIT/GE/ENG/88S-1

A MULTIPLE-VALUED LOGIC SYSTEM FOR CIRCUIT EXTRACTION TO VHDL 1076-1987

THESIS

Michael A. Dukes
Captain, USA
AFIT/GE/ENG/88S-1

Approved for public release; distribution unlimited

AFIT/GE/ENG/88S-1

A MULTIPLE-VALUED LOGIC SYSTEM FOR CIRCUIT EXTRACTION TO VHDL 1076-1987

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

Michael A. Dukes, B.S.

Captain, USA

September 1988

ACKNOWLEDGMENTS

I would like to express my sincerest appreciation to my loving wife, [redacted] for her undaunted patience, support, and endurance during the course of my graduate program and research. Her cheerful and devoted attention to our home provided me the freedom and strength to do this work.

I wish to thank Major Joseph DeGroat for his guidance and willingness to offer me the greatest freedom during my research. His philosophy on mentor and student relations allowed for the fullest development of my research skills. I also wish to thank Major DeGroat, Professor Frank Brown, and Captain Bruce George for reviewing my thesis and adding their highly informative and constructive critiques on my research and writing. I am also grateful to Captain Keith Jones for our insightful midnight chats in the VLSI lab.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Multiple-valued logic is a topic of concern for modeling standards in the VHSIC Hardware Description Language, VHDL 1076-1987. With the various forms of layout styles in MOS devices, there exist different strengths of electrical signals propagated throughout a circuit. Additionally, logic extraction to VHDL of VLSI layout designs may contain leftover transistors that must be modeled correctly in VHDL. A multiple-valued logic system can adequately model signals with different strengths as well as conflicts between signal values. Once a multiple-valued logic system is defined, a logic extraction system may then produce VHDL for hardware component representations down to the transistor level. The goal of this thesis is to present a ten-level multiple-valued logic system and provide a Prolog-based logic extraction tool for generation of VHDL from a transistor netlist. The Prolog-based logic extraction system will also provide groundwork for further research in the area of formal verification with VHDL. Various tools using symbolic representations and multiple-valued logic are essential in a CAD environment where logic extraction from layout to VHDL is incorporated into validation and verification.

COMMON ABBREVIATIONS

Abbreviation	Explanation
AND	Operation of logical conjunction
CMOS	Complementary Metal-Oxide-Semiconductor
CAD	Computer Aided Design
cif	Caltech Intermediate Format
DoD	Department of Defense
esim	Switch level simulator
GND	Ground or Zero Volts
HDL	Hardware Description Language
NAND	Complemented operation of logical conjunction (nonassociative)
NOR	Complemented operation of logical disjunction (nonassociative)
OR	Operation of logical disjunction
SIF	Symbolic Intermediate Form
sim	Transistor netlist generated by mextra
STOVE_C	Sim to VHDL extraction tool using C
STOVE_P	Sim to VHDL extraction tool using Prolog
Vdd	Supplied Voltage, High or +5 Volts
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
VLSI	Very Large-scale Integration

A MULTIPLE-VALUED LOGIC SYSTEM FOR CIRCUIT EXTRACTION TO VHDL 1076-1987

I. INTRODUCTION

Background

The VHSIC Hardware Description Language, IEEE standard VHDL 1076-1987, is an important part of the Air Force Institute of Technology CAD environment. A conceptual view of a general CAD environment is presented in Figure 1. The initial concept of a hardware design begins as a VHDL behavioral specification. The VHDL behavioral specification of the circuit may be decomposed into a VHDL structural description through a schematic capture system. A structural description is then converted to a layout description in an intermediate layout representation through a compilation tool or manual layout. The layout is translated into a mask level description for transmittal to a Silicon foundry service for fabrication. Tools also exist to perform a switch extraction from the mask layer description. The transistor netlist description may then be used for switch level simulation or for extraction to a gate level representation in VHDL.

In order to provide a concrete example of how multiple-valued logic and extraction to VHDL may be accomplished, the CAD environment at AFIT, shown in Figure 2, will be used for discussion. Magic is a manual layout tool from the Berkeley Distribution of Design Tools (California 1986). The mask level

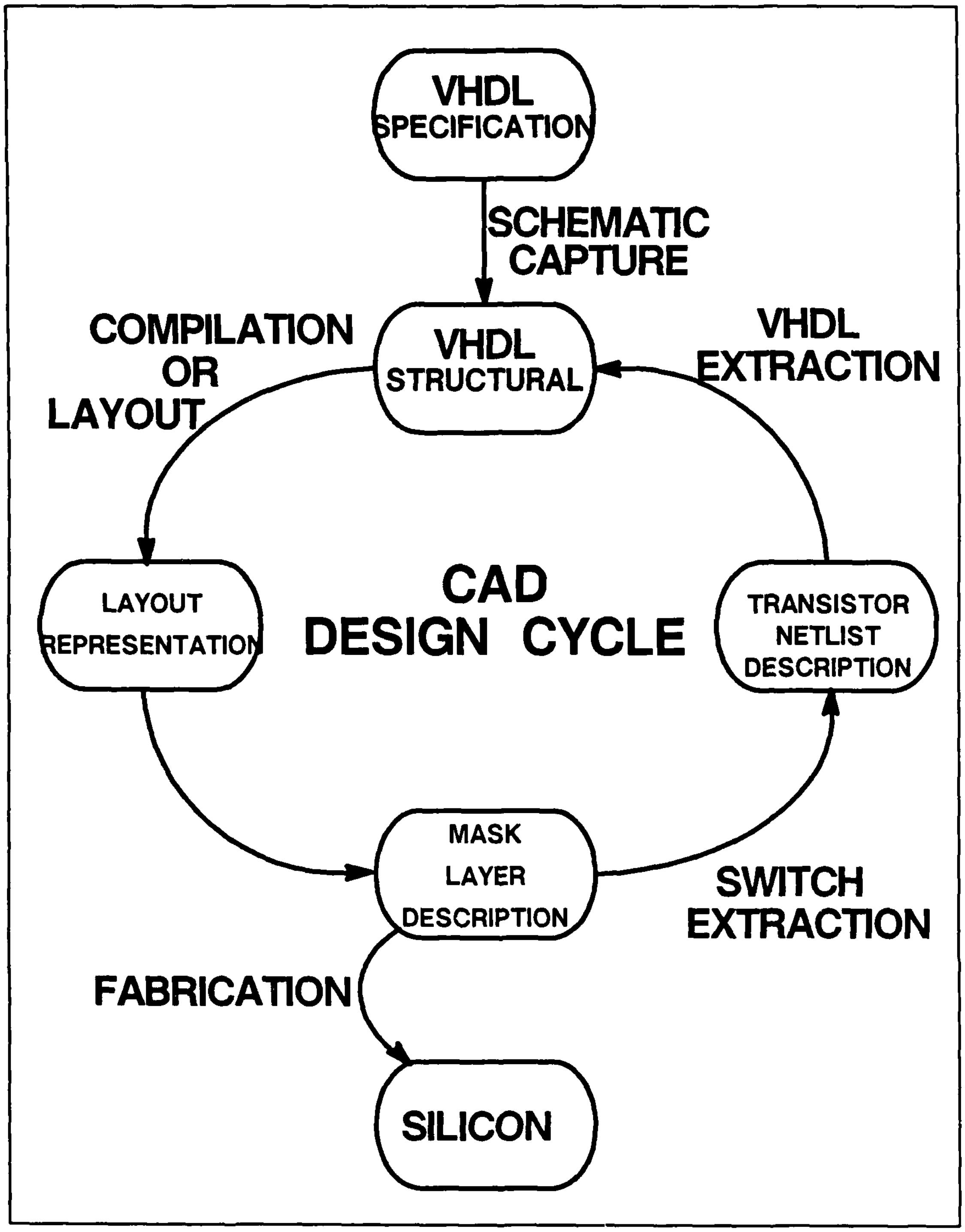


Figure 1. General Description of a CAD Environment.

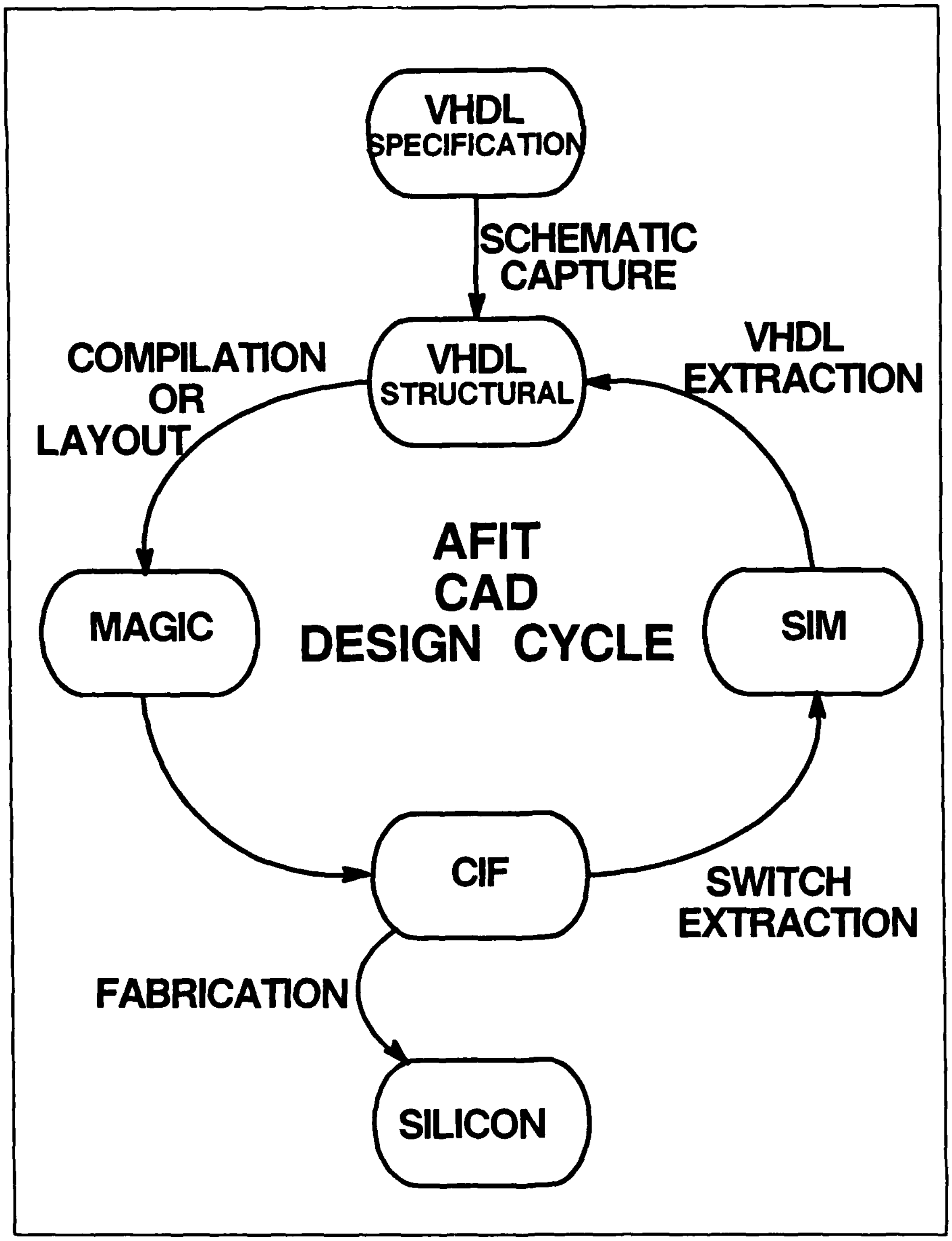


Figure 2. AFIT CAD Environment.

description generated in magic is stored and transmitted in the CalTech Intermediate Format (CIF). The CIF format is sent to the MOS Implementation Service (MOSIS) for fabrication of VLSI components. The sim transistor netlist representation is extracted from CIF by a tool called mextra. The sim is an input form to a switch level simulator called esim, also found in the Berkeley Distribution of Design Tools.

Switch-level simulators are commonly employed to simulate designs that have been generated using a manual layout tool. The esim simulator is an example of a switch-level simulator that uses a sim transistor netlist representation generated from a magic VLSI layout description. As a result, the simulation of the design is performed at a switch-level representation of the transistor through a state model. In this state model, the values on all nodes must converge to a "steady-state" condition before generating a report of the new values on all nodes. A common design practice is the use of feedback loops in circuit configurations. These feedback loops cause problems within switch-level simulators since the temporal aspect of the design is not considered within the simulator. Effectively, feedback loops prevent values on nodes from converging. The solution for simulation in switch-level simulators is either to convolute the switch-level representation of the circuit through software or to modify the layout design.

Modification of the switch-level representation through software leads to perturbations of the modeled circuit that may not truly represent the original layout design. Modification of the design layout to accommodate the esim simulator also leads to a design that is difficult to interpret by other engineers and to possible unforeseen side effects, e.g., race conditions and susceptibility to cross-talk.

A SIM-to-VHDL (STOVE_C) CAD tool is under development at AFIT. Its purpose is to provide VHDL from circuit designs that have been generated in magic and translated to the CIF format. The percentage of extraction success varies from chip to chip. This is due to the multitude of possible transistor configurations that may exist within a chip design. Therefore, transistors must have a functional representation in VHDL as part of the full design. Furthermore, testing of components requires the use of a transistor model for the circuit under test. However, no requirement exists to represent the entire unit under test at the transistor level. Portions of a unit under test may be represented at the transistor level while maintaining a higher level representation of the nontested portion of the circuit. Esim and other switch-level simulators do not allow abstraction of hardware above the transistor level. Since VHDL allows for abstraction to a high level, the efficiency of the simulation is improved.

Efficiency of the VHDL simulator is realized through abstract behavioral representations of large groups of transistors while maintaining low-level transistor representations of the circuitry under test. VHDL's treatment of timing overcomes the convergence problems within esim. Since VHDL is based on a timing model, the testing phase of the CAD cycle can be enhanced by a simulator that not only produces the signal values to be observed, but also provides information on when the signal value changes should occur.

Problem Statement

There are several problems with validation of VLSI designs.

(1) A switch-level simulator does not accurately simulate timing models and several different forms of VLSI layout style.

(2) The default two-valued logic system within VHDL is insufficient for direct simulation of VLSI layout designs.

(3) The STOVE_C tool that does exist for logic extraction is difficult to modify.

The purpose of this thesis is to develop a logic extraction system and a multiple-valued logic system for VHDL. Multiple-valued logic systems are necessary within the realm of logic extraction, since not all VLSI layout designs are directly extractable to gate logic representation. A multiple-valued logic system will allow for the inclusion of transistors in the normal gate-level VHDL simulation model.

Scope

This thesis is bounded both in the type of problems to which it applies and in the scope of the solution it presents.

Scope of the Problem. The problem space considered by this thesis is in the realm of custom MOS layout design that can be extracted to a transistor level netlist form. Other systems that use semi-custom VLSI design, where the "black-box" cells are simply wired together, are not considered. Furthermore, other types of technologies that do not exhibit the behavior of MOS VLSI designs are also not considered. Such technologies not included in this thesis may be classified as bipolar.

Since capacitive sharing is also a part of MOS VLSI design, a system for handling the propagation of these values must also be incorporated into a multiple-valued logic system. Therefore, it will be important to provide a proper treatment for bus lines and domino forms of MOS VLSI design. However, analog circuits, to include switched capacitor circuits, will not be addressed.

The multiple-valued logic system must also work within the VHDL simulation model. As such, the temporal aspect and simulation cycle of VHDL simulation will provide the basis for the multiple-valued logic models that are developed.

Scope of the Solution. Since the application of the software tools developed in this thesis is aimed at closing the design loop in the AFIT CAD environment, the form of the solution will concentrate on custom MOS VLSI design. Most of the forms of layout styles encountered will generally involve derivatives of the forms found in (Weste and Eshraghian 1985). The extraction system will start after the generation of the transistor netlist from Mextra and end with a VHDL 1076-1987 final output. The intermediate representation developed will be in Prolog clause form and may provide the basis for a Symbolic Intermediate Form (SIF).

Summary of Current Knowledge

Very High Speed Integrated Circuit (VHSIC) technology has been a topic of concern within the Department of Defense since the late 1970's (Barna 1981:2). The technology regarding this area has advanced, but the design methodologies or hardware description languages have not. The VHSIC Hardware Description Language (VHDL) has evolved over the past several years as a whole design process in response to the gap between VHSIC technology and hardware description languages.

One of the major goals of VHDL is the insertion of the latest technology into new and existing systems. This goal is realized through VHDL's support of design, documentation, and efficient simulation (Shahdad and others 1985:94). VHDL supports simulation from gate-level to system-level descriptions. Intermetrics has developed a VHDL support environment consisting of an analyzer, design library, library manager, simulator, and reverse analyzer. The VHDL V7.2 Intermetrics version of the simulator was written in Ada to run on a VAX system under the VMS operating system. Until 1987, the complete semantics of VHDL was not supported by the Intermetrics BUILD2 simulator. The Intermetrics BUILD3 version of the simulator supports all of the features of VHDL V7.2, but is still very slow. In December, 1987, VHDL 1076-1987 was introduced by the IEEE as a standard HDL. Intermetrics produced a working analyzer and simulator in July 1988.

An important part of a complete hardware description language is a simulator. Since the number of effective devices on an IC chip has increased beyond the reasonable application of any testing, simulation of the device seems to be the only valid means of evaluation. However, if a simulator cannot supply an adequate report of the validity of a device within a reasonable period of time, it is useless. In order to write an efficient simulator or other CAD tool, it is important to have an understanding of the hardware description language and the target system for its use.

In order to gain some background knowledge of VHDL it is important to review important literature produced over the past several years. The April 1986 issue of IEEE Design & Test contains a considerable collection of articles. Further information is found within the VHDL User's Manual. A VHDL lecture series produced by the Air Force Institute of Technology (AFIT) supplements the Intermetrics tutorial.

Most of the articles contained within the IEEE Design and Test magazine pertain to background knowledge of VHDL. Specifically, the articles (Aylor 1986; Dewey and Gadient 1986; Gilman 1986; Lipsett and others 1986; Lowenstein and Winter 1986; Nash and Saunders 1986) contain information on a comparison of VHDL to other hardware description languages, syntactical features within VHDL, hardware testing with VHDL, and motivation for VHDL.

VHDL was compared to seven other hardware description languages available in industry and universities. The criteria of the examination were scope of hardware design, management of design, timing description, architectural description, interface description, design environment, and language extensibility. The scope of hardware design is concerned with the level of the design from the gate to the system level. The management of the design is encompassed by the hierarchy, modularity, and control aspects of the design process. Timing descriptions are most important for specifying inertial timing, transport timing, and propagation delay. The architectural aspects concern the algorithms, structures, parallelism, separation of control and data, and components. Interface descriptions should be defined, strongly typed, and explicitly stated at all levels. The design environment concerns identification of the external physical influences on the functional component. Finally, the language extensibility addresses the description language's ability to maintain user-defined types, design tool support, multiple technologies, and multiple methodologies. Overall, VHDL incorporates the most important aspects of these areas (Aylor 1986:17-27).

An overview of the design environment and VHDL syntax was provided (Gilman 1986; Lipsett and others 1986). The features of the design environment described were the VHDL analyzer, design library, library manager, and simulator. The hierarchical nature of the VHDL language syntax from design entity down to the signal assignment statement was reviewed. Aspects of the design language pertaining to separation of signal types from data types and specification of timing constraints were discussed with specific references to the VHDL syntax. An example was provided on how the majority of the VHDL syntax incorporated constructs of Ada into a hardware description language. As software development with Ada uses a software support environment and software language, the VHDL encompasses the support environment described and the VHDL language.

The problems faced by today's gap between technology and description languages was thoroughly covered by Al Dewey and Anthony Gadient (Dewey and Gadient 1986). The industry has been relatively incapable of producing documentation that is current, detailed, and accurate. Technology has enabled the incorporation of VHSIC structures on chips at a magnitude beyond currently available description languages. Additionally, military systems have usually been application specific. "Off-the-shelf" components do not exist for ready use in these systems. VHDL is supposed to decrease the design time and cost of specific application components. Furthermore, it should provide for ease in second sourcing and protect proprietary design system elements (Dewey and Gadient 1986:12-16).

Testing of design components is normally performed differently at specific levels of the design (i.e., whole system, printed circuit board, chip, etc.). However, the behavior and mutual effects of separate subcomponents need to be incorporated into the test process. Likewise, different types of tests exist for each phase of the device life-cycle. These life-cycle phases are design/development, production, and maintenance. VHDL accounts for the electrical, physical, and environmental factors necessary for constructing tests to be applied within the different phases of the product life cycle (Lowenstein and Winter 1986:48-53).

The VHDL User's Manual provides a tutorial on the syntax of VHDL V7.2. Chapters 2, 3, 5, 6, and 7 provide information on how the constructs of VHDL are used within the simulator portion of the VHDL design environment. The description of transactions and how they evolve into events as part of an event-driven simulation is an important part of the discrete simulation process. Modeling hardware structures, asynchronous circuits, synchronous circuits, and complex behaviors are some of the topics important to simulation of hardware design. The use of sequential signal assignment statements for specification of concurrent events is important for the simultaneous specification of multiple drivers on a bus. The cycle of the simulation from time zero to the cessation of further transactions is also presented (Intermetrics 1985).

The VHDL video lecture series was produced and assembled at AFIT. The lecture series is currently archived by the Design Group, MicroElectronics Branch, Air Force Wright Aeronautical Laboratory. It is a tutorial to complement the Intermetrics VHDL User's Manual. Concrete examples are presented in conjunction with a comprehensive view of the syntax and modeling aspects of the VHDL language (Linderman 1986; DeGroat 1986).

Assumptions

The logic extraction tool and the ten-level multiple-valued logic system is to be developed for CMOS design within the AFIT CAD Design Cycle. Therefore, it is assumed that a transistor netlist representation of a chip design will be generated in sim format. The sim format will be provided through a software tool called Mextra even though most any netlist representation for CMOS circuits should be acceptable. Furthermore, the typical CMOS designs employed will largely reflect those from Weste and Eshraghian.

The presentation of this thesis also assumes background knowledge in several areas. The reader should be familiar with VLSI layout design in a CAD environment. Some knowledge of Prolog and C would also be helpful for interpreting the examples and portions of the extraction code. A working knowledge of VHDL is not necessary; however, an appreciation of the various hardware simulator modeling stratagems is critical to the understanding of the issues raised in this thesis. A background in abstract algebra and predicate calculus would also assist the reader. A brief list of sources that would provide some cursory knowledge is provided in Appendix A.

Approach

This research entailed two efforts that could be pursued in parallel. The first was to analyze the behavior of CMOS circuits and adopt an appropriate algebra as a basis for a VHDL multiple-valued logic system. The second involved the analysis and development of methods for Prolog modeling of VLSI designs in order to generate a methodology for logic extraction and an approach to the generation of VHDL.

In the first effort, involving multiple-valued logic, several multiple-valued logic systems were surveyed. Afterwards, an algebra was developed that allows for a simulation model of CMOS design that is more accurate than the current two-valued logic system. Two methods for incorporating the multiple-valued logic system into VHDL were studied, one of which was adopted.

The second effort concerned sim-to-VHDL-extraction using a Prolog symbolic representation. Several techniques for modeling VLSI designs in Prolog were reviewed. A pattern recognition methodology was adopted that modeled typical human identification of transistor patterns for extraction to higher level components. A symbolic intermediate form in Prolog clause form was developed to accommodate the extraction and VHDL generation tools.

Once the multiple-valued logic system was written in VHDL and the Prolog based logic extraction system, STOVE_P, was generating VHDL using "BIT" types for signals, the multiple-valued logic system could be incorporated in the STOVE_P system to generate VHDL for any VLSI design in Magic. The final form produced VHDL V7.2 and VHDL 1076-1987.

Materials and Equipment

Since this effort will involve production of VHDL, a VHDL analyzer and simulator will be required. Furthermore, a Prolog interpreter will be necessary for development of the extraction tools. Various cells laid out using the Berkeley VLSI designtool, (California 1986) Magic, and translated to a netlist representation through Mextra will be needed to generate the necessary input files for use in this tool development.

The VHDL environment, Prolog, and Berkeley VLSI design tools run on VAX/VMS, VAX Unix, Elxsi Unix, and SUN Unix. A VAX 8800 and VAX 11/785 running VMS were used at AFWAL for STOVE_P and VHDL. A VAX 11/785 running BSD Unix V4.3 was used at AFIT for initial design of the Prolog extraction tools. An Elxsi 6400 running BSD Unix V4.2 at AFIT was used for Mextra. Finally, a SUN 3 running Unix V4.2, release V3.2, was used at AFIT for the layout of several VLSI designs.

Sequence of Presentation

Chapter 1 has provided some background on VHDL in the AFIT CAD Environment and AFIT VHDL Environment. The problem statement was presented as well as the scope of the problem and solution. An approach to the solution was summarized.

Chapter 2 is a review of the current approaches that exist in the academic community for multiple-valued logic systems and in the AFIT CAD Environment for logic extraction. Three multiple-valued logic systems are summarized as well as the work on STOVE_C.

Chapter 3 contains a detailed problem analysis concerning logic extraction and simulation modeling. The problem of using the most appropriate software approach is discussed. Several issues concerning simulation modeling of initialization, driver strengths, and logic flow to include their effects are studied.

Chapter 4 details the software design to accomplish both logic extraction and VHDL modeling with multiple-valued logic. The software process from the cif representation to simulatable VHDL is presented. An algebra for a ten-level multiple-valued logic system is presented.

Chapter 5 discusses how the ten-level multiple-valued logic system was incorporated into VHDL. The implementation of the logic extraction system is also explained.

Chapter 6 details the extraction and simulation of several VLSI designs from the AFIT CAD Environment. A multiplier chip is used to demonstrate the parallel processing capability of the logic extraction system.

Finally, Chapter 7 details the findings from logic extraction and simulation modeling with a ten-level multiple-valued logic system. The relevant application of the research to further thesis and doctoral work is also discussed.

II. REVIEW OF CURRENT APPROACHES

Introduction

The purpose of this chapter is to summarize approaches to multiple-valued logic systems and review the development of STOVE_C. Analysis of the different views toward multiple-valued logic systems will help identify some of the problems in hardware simulation of transistor designs, specifically, problems with initialization, weighting of signal values, and detection of hazard conditions. A logic extraction tool, STOVE_C, will also be examined to determine what benefits were realized and what problems were encountered.

Multiple-Valued Logic Systems

There are several multiple-valued logic systems in existence that address modeling of MOS transistors. MOSSIM is one that uses a multiple-valued logic system based on logical 0, 1, and X (unknown) (Bryant, 1981: 786). The effects that these logic values cause on the circuit pertain to the gate level and lower levels. Table 1 shows the values as they are determined by the gate type of transistor. As a result of using a transistor-level representation of the circuit, bidirectional flow may be easily modeled. Gate-level simulators only handle unidirectional flow of signal values and are incapable of handling a typical transmission gate representation unless some premise is made as to the direction of logic flow. In addition, devices that use high impedance output are not simulatable under a two-valued logic system.

Table 1. Logic Values and Their Effects (Bryant, 1981)

n-type		p-type	
gate state	effect	gate state	effect
0	open	0	closed
1	closed	1	open
X	unknown	X	unknown

Though Bryant's MOSSIM simulator addressed the initialization problem inherent with two value logic by using a logical X state, the problems with capacitive strength levels on signals were not considered. In a later paper (Bryant, 1984), Bryant attempted a solution within the MOSSIM simulator, MOSSIM II. In this case the circuit network was considered as a collection of nodes and transistors, where the nodes interconnect the transistors (Bryant, 1984: 161). In this model, each node was weighted in relation to its expected capacitance and each transistor was weighted based on its relative conductance (Bryant, 1984: 162). The network would then be evaluated based upon matrix analysis quite similar to circuit-level simulators, e.g., Spice. This type of circuit analysis was far from the evaluation methodology of VHDL, and therefore, impractical for consideration.

To compensate for the problems of dynamic and static charges that can occur on devices in CMOS, additional levels of logic values were addressed in another paper. One solution (Watanabeand others, 1980) involved the use of dynamic and static logic symbols. The logic values are shown in Table 2. The logical 1 and 0 are derived from signals that are driven by a source, i.e., Vdd and GND. The logical X is indicative of an uninitialized state. The logical *1 is commonly the result of capacitive charge which may exist on a given node. The logical *0 is the absence of charge. Logical Z exists on a node where the charge due to a logical *1 or *0 has dissipated after some dynamic holding time. Static charges have precedence over dynamic charges. Finally, logical E represents a condition where some electrical rule has been violated. Typically, this occurs when two static charges are competing for the same node, as might occur when Vdd and GND are shorted together through a path of low resistance.

Table 2. Seven Value Logic (Watanabe and others, 1980:942)

Symbol	Static	Dynamic
Logical High	1	*1
Logical Low	0	*0
Unknown	X	Z
Error	E	E

Several aspects of circuit behavior are modeled through this logic value system. Uninitialized values are unknown, logical X. Static and dynamic charges are modeled through separate values. Hazard conditions between static charges are reported through E. Knowledge of the direction of circuit flow is not required for this model. Thus, this aspect of modeling allows for bidirectional circuits. The analysis does not encompass the problems of large resistances or resolution of large capacitive signals over small capacitive signals.

Ullman's multiple-valued logic system is applicable to both NMOS and CMOS models. The system is based on a Hasse diagram, therefore, there is a base state where nothing has been initialized. Table 3 is a description of the multiple-valued logic system used by Ullman (Ullman, 1984: 412). The logical X, when used with the strength levels denotes an unknown or hazard condition. The 0 represents GND and 1 represents Vdd. The strength levels shown in Table 3 are listed in strength precedence. Thus, a node with logic value SC0 would be overridden by logic value W1 and assigned a new logic value of W1. Likewise, for a node with a logic value of SC0, competition with a logic value of SC1 would render a logical SCX.

Table 3. Ullman's Multiple-Valued Logic System

Strength	Symbol		
Driven	D0	D1	DX
Weak	W0	W1	WX
Super Charged	SC0	SC1	SCX
Charged	C0	C1	CX
Bottom		B	

The bottom logic value is used to represent an uninitialized state. All nodes are originally assigned logical values of B until a new value can be computed for the node based upon the simulation model used. The relative basis of each strength level from Table 3 is determined by the circuit. For a strength of D, the source is typically Vdd and GND. The strength of W results from either a driven or weak source passing through a large resistive transistor. The SC strength level is the result of a circuit with a large capacitance isolated from a driven or weak source. Finally, the C strength is described as a small capacitive circuit isolated from a driven, weak, or super-charged circuit (Ullman, 1984).

Ullman describes several simulation methodologies using this multiple-valued logic system. Upon creation of the circuit, all nodes are assigned a logic value of B. Afterwards, all nodes that are connected directly to Vdd and

GND are assigned logic values of D1 and D0 respectively. For those areas that are not discernible, but are driven, the logic value of DX is assigned. Nodes that are isolated from driven sources by large resistive loads are assigned the corresponding weak logic value. For large capacitive sources, a logic value of SCX is assigned. Finally, for regular capacitive nodes, a logic value of CX is designated. After the initialization sequence has been executed, the circuitis are then iteratively evaluated using the relations described above (Ullman, 1984:420–421).

The algorithm described in (Ullman, 1984:408–425) was largely descriptive of NMOS applications. Further refinement can be made for most CMOS applications. The description of the modifications will be covered in a later section.

STOVE_C

The STOVE_C project, originated by Captain Richard Linderman, has been in development for the past 18 months (Gallagher 1987:139–151). The major goal of STOVE_C was to extract VHDL from a sim transistor netlist representation of a VLSI design. A secondary goal of this routine was to extract descriptions in the CHIEFS format for fault tolerant design testing. The extraction strategy was to operate on the transistor file from sim in stages. Each stage represented a further level of component extraction. The first stage would be to extract inverters, transmission gates, and clocked inverters; the second stage of this process would then involve constructing other logic gates and flip–flops that were constructed from the components of the first stage and leftover transistors (Gallagher 1987:139–140).

The input to STOVE_C is the sim file transistor netlist. For each line, there exists a predefined field of one character in the first position. If a recognized value for this field does not exist, then an error message is sent to the terminal. Currently, lines in the sim file containing capacitivei nformation are ignored (Gallagher 1987:140-141).

As status information, STOVE_C provides feedback on errors, number of nodes processed, number of transistors of a given type, number of specific transistor patterns found, and a percentage of unextracted transistors. A list of unextracted transistors is provided in a separate file. This list provides the designer information that may help to analyze possible layout errors. Finally, a third file is created that contains all of the extracted higher level components in a format that may be input to CHIEFS. Even though some VHDL is produced, the VHDL was never analyzed and simulated for correctness as a simulation model (Gallagher 1987:142-144).

Recognition of various design components is dependent upon the naming conventions in use at AFIT. Therefore, use of IZ, OZ, and BZ, to list a few, assists the extraction routine. Because of this labeling dependence, designs that do not provide labeling of nodes in the same manner complicate the extraction methodology (Gallagher 1987:143). Therefore, some modification would have to be performed on STOVE_C to integrate it into other CAD environments for VLSI layout, since it would be far more difficult to completely alter a CAD environment for the integration of one tool.

Most of the effort in developing STOVE_C centered on the incorporation of data structures and algorithms that would process the logic extraction by the fastest means possible. The structure to define a transistor incorporates the use of 13 fields for pointing to transistors, higher level components, and other identifying information about the transistor. Further structures are used to define higher level components and gates (Gallagher 1987:144–145). The extraction process and data structures were developed for a signal processor environment. No consideration was made for supporting parallel processing.

Other tools have been incorporated into STOVE_C. One tool modifies the transistor netlist to overcome a problem that esim has with feedback. The nofeed option for eliminating feedbacks deletes the feedback inverters in master–slave flip–flops and D flip–flops so that esim can then simulate the design. As discussed later in this thesis, the problem of feedbacks in esim is due to its state–model representation. Another tool, fixrom, also modifies the transistor netlist for esim. The problem with ROMs in esim is in the operation of the senseamp. A portion of the ROM cell is removed so as to eliminate contention over the proper value of a signal (Gallagher 1987:147). In essence, these tools modify the design to accommodate the simulator. However, the nofeed tool does not eliminate all of the possible feedback loops that are used in VLSI design. A ring oscillator is an example of a feedback design that will not simulate in esim.

Conclusions

A methodology for logic extraction, STOVE_C, does exist. The system employed in STOVE_C makes extensive use of fast algorithms to efficiently perform logic extraction. However, there are still some issues that remain to be resolved. Inclusion of additional component descriptions requires the production of additional C functions and data structures. This problem may hinder the incorporation of new design styles into the AFIT CAD environment. No real intermediate form exists that can be used for formal verification. Esim is still used as the primary target for simulating designs from STOVE_C rather than looking to VHDL as the primary simulation tool. The structure of STOVE_C is not adaptable to a parallel processing environment.

Several important points were iterated from one multiple-valued logic system to the next. Bryant discussed the problem of properly performing initialization. Watanabe and Ullman were concerned with differentiation between the relative strength levels of signals based upon the implemented circuit. Finally, Watanabe's methodology for specifically identifying hazardous conditions as a means to detect design flaws was identified as a necessity. The following chapter will analyze the problems in simulation and logic extraction with respect to the approaches reviewed in this chapter.

III. PROBLEM ANALYSIS

Introduction

The previous chapter explored approaches to logic extraction and multiple-valued logic systems. Initialization, signal strength levels, and detection of hazard conditions were enumerated as necessary elements of a simulator that models MOS behavior. However, the SIMULATOR_STANDARD package of VHDL does not provide direct support for modeling MOS transistors. Therefore, a logic extraction system is not completely supported using the SIMULATOR_STANDARD package of VHDL.

A number of problems with STOVE_C and its implementation have been identified during the course of this thesis effort. An intermediate form is not a requirement for STOVE_C; however, an intermediate form for symbolic representation may become an important part of formal verification. Another problem identified with STOVE_C is that it does not produce simulatable VHDL. Furthermore, Prolog has been identified as a better language for rapid prototyping solutions than C. C is also unforgiving when errors occur in logic, whereas Prolog's clause form is a predicate calculus form of logic. What can be expressed in a few lines of Prolog require more lines of Lisp or C, thus a Prolog solution will decrease the burden of ensuring code correctness on the programmer. The purpose of this chapter is to further detail the points listed above.

Simulation Problems

The issues considered in this section are initialization, conflicts between power and ground, driver strength levels, and bidirectional flow. A two-valued logic system inhibits the simulator's potential to identify and model these areas.

Additionally, logic extraction is increasing in importance in the AFIT CAD environment. Gate level treatment of circuits alone is not suitable for circuit extraction. The following discussion will cover these areas in more detail.

Initialization. The two-valued logic system of 0 and 1 does not account for uninitialized nodes within a circuit. Though it is possible to establish *a priori* values of 0 and 1 on nodes throughout a circuit, nodes that are never stimulated during the simulation cannot be located. Use of an uninitialized state allows for identification of nodes within the circuit that do not initialize to a known state. It also indicates isolated circuitry that might require attention to handle conditions such as "power up," "reset," or "power up test." This state is necessary to provide accurate information to the designer.

Conflicts Between 1 and 0. The X state designates nodes that are driven 0 and 1 simultaneously. As such, the X state also indicates areas of a circuit where power and ground may be "shorted" together due to errors in circuit layout. In this manner, hazard conditions may be identified throughout the circuit where X states exist.

Driver Strength Level. Two-valued logic does not differentiate between strength levels on a signal or a node. Bus resolution functions are used to resolve conflicts between drivers on a signal; however, without specific knowledge of the circuit, the bus resolution function performs assignment irrespective of the relative strength of the drivers. Within logic extraction, there is insufficient knowledge of the strengths of various drivers for straightforward generation of an accurate bus resolution function.

Bidirectional Flow. Logic-level simulators consider logical flow in one direction, e.g., a gate receives inputs on one set of lines and generates output on a separate line. For transistors and transmission gates, the input and output may be observed on any of the interconnecting lines. Within the transistor level representation of a logic gate, current flows in one direction; however, logical values may flow in two directions as in Figure 3.

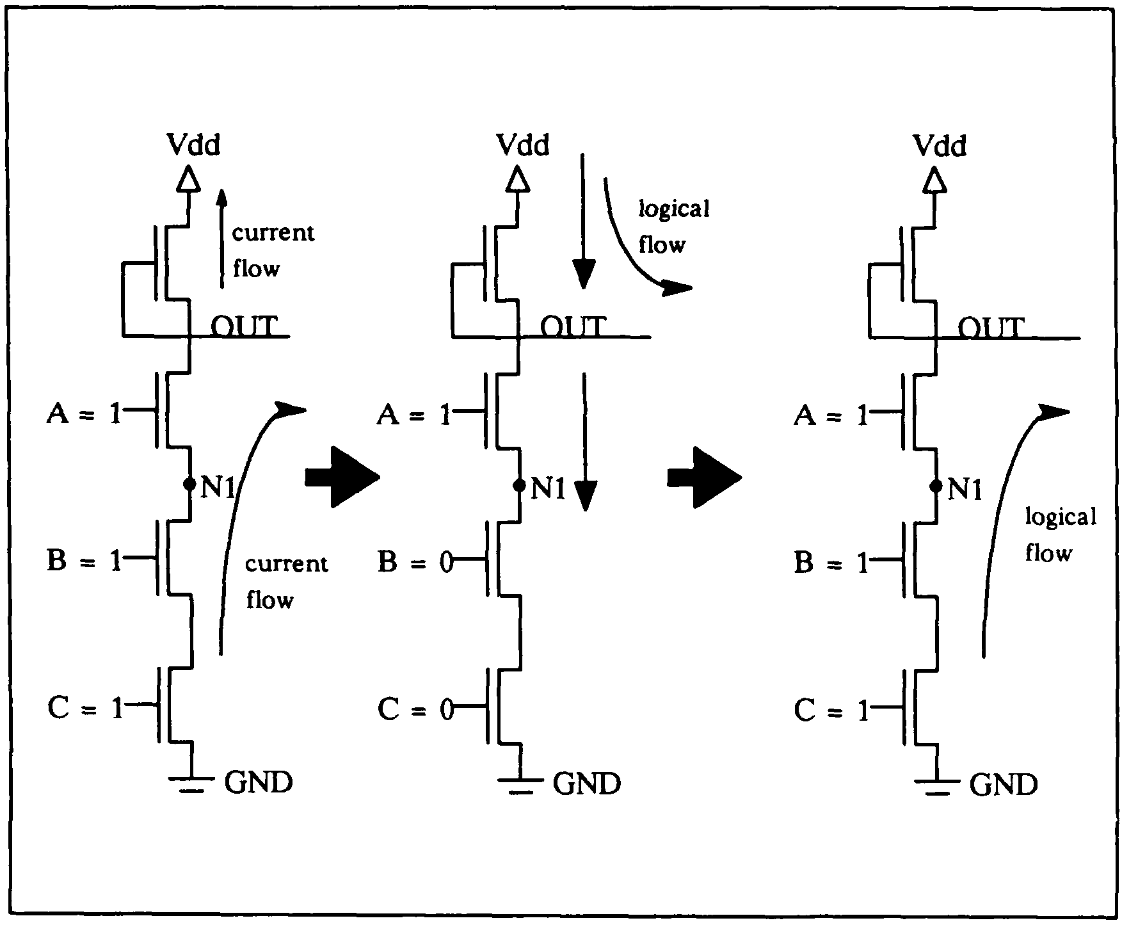


Figure 3. Logical Value Flow in an nMOS NAND Gate.

In the NAND gate of Figure 3, electrons flow from GND to OUT when A = B = C = 1 is met; however, when the condition of A = 1and B = C = 0 occurs, the logical 1 value from Vdd is propagated through the pullup transistor to OUT and N1. In this case, logical values are propagated downward through the circuit. When B = C = 1 occurs, the stronger 0 value from GND propagates upward through N1 to change the value of OUT from a weaker 1 to a driven 0 value. In this case, logical values have propagated in two directions through N1 and the transistor whose gate is A. A transistor representation in a simulation must be treated as a bidirectional device even when the actual current flow may be deduced from the transistor configuration.

Integration with AFIT CAD Environment. The primary goal of a ten-level multiple-valued logic system is to provide the basis for complete extraction of VHSIC class components specified in CIF format. Secondary effects within the AFIT CAD environment include support for observability in testing, formal verification, and VHDL respecification. Some chip designs in the AFIT CAD environment were previously designed without a VHDL specification. Some of these chip designs contain in excess of 100,000 transistors. Even with the complete custom VLSI cell library, manual logic extraction of VHSIC class components present a formidable task.

Software Specification

When choosing the appropriate software support environment for implementation, several issues must be considered. One classic controversy in software design is "efficiency vs readability." Readability of code also influences code rapid prototyping, development, and maintenance. With VHSIC class chips realizing 100,000 upwards to 500,000 transistors currently fabricatable, the issue of efficiency cannot be completely disregarded for the sake of readability.

Furthermore, one goal of this thesis is to analyze the feasibility of symbolic representation for VLSI design. A form of symbolic representation will be important in the realm of formal verification. Therefore, the language should be readable, reasonably efficient, and support symbolic representation of VLSI design.

Figure 4 is an example of the minimal C code data structure that could be required to represent a p-type transistor and recognize the fact that the drain and source are interchangeable. Within the structure, fields are identified for pointers to other possible transistor structures as well as the appropriate node label. The data structure conveys no further information as to how it will be used. Additional C coded algorithms must be provided to perform searching and linked list manipulation.

```
struct Trans
    {
    NODE *gate,*source,*drain;
    TRANS *gnext,*snext,*dnext;
    int length,width,Txpos,Typos;
    char kind,use;
    TRANS *team;
    int id_num;
    };
```

Figure 4. C Code Data Structure for Defining a Transistor in STOVE_C (Gallagher 1987:147).

Figure 5 is an example of an equivalent means of specifying transistors in Prolog. If the first letter of a label in Prolog is upper case, then a variable is specified. First letters in lower-case specify a known value or atom. The Prolog example conveys several items of information. If an entry specifying a p-type transistor exists on the database, then its drain and source are interchangeable. Furthermore, the Prolog code also specifies relations on the database of transistors. Given any known value for the gate, drain, or source, the Prolog code shown in Figure 5 will return all possible instantiations that match the prerequisite. To accomplish this same task in C code would require the use of additional C code for each type of search, whereas the different searches are implied by the Prolog code in Figure 5.

```
ptrans(G,D,S) :-
    p(G,D,S).
ptrans(G,D,S) :-
    p(G,S,D).
```

Figure 5. Prolog Structure to Define a Transistor.

The point demonstrated in Figures 4 and 5 is that for the pattern matching types of problems required to perform logic extraction, a few lines of Prolog code can perform the same operation that would require numerous lines of C code. However, the pattern of the search performed in Prolog is transparent to the user as it is inherent in the language. It is possible to construct near optimal searching sequences in C that are more efficient when compared to specific Prolog environments. Therefore, an analysis of the Prolog approach for efficiency should be considered only after all requirements have been specified and an enhanced Prolog support environment is available.

Conclusions

Two main issues were presented. In order to portray a working model of CMOS design that includes transistors, it will be important to develop a multiple-valued logic system that handles initialization, bidirectional logic flow, and the different driver strength levels in MOS design. Even though using C code for implementing a logic extraction methodology would be highly efficient, its use in rapid prototyping and modification may impede the necessary requirements analysis concerning symbolic representation of MOS circuits. Therefore, Prolog was a primary candidate for constructing the logic extraction system. The important point within this software approach was to use a language that is best suited to the problem and not to advocate the strict use of any one language.

IV. ACTUAL DESIGN

Introduction

In this chapter, the design of the software for logic extraction and multiple-valued logic will be discussed. Figure 6 shows a system to extract logic and produce VHDL. The cif file representation of the VLSI design layout is passed through a software routine called upper. The upper routine normalizes all labels to upper case letters only. Then the resulting file is passed through mextra to produce the transistor netlist file in sim format. Afterwards, the sim file is then converted from sim to Prolog format using a routine called sim2pro. At this point, a Prolog routine called trans is used to iteratively perform extraction on the transistor and component databases in Prolog. The fully extracted Prolog form is then translated to VHDL using a routine called pro2vhdl. Pro2vhdl contains the necessary component instantiations and ten-level multiple-valued logic to support the simulation model.

Label Conversion

Mextra and Prolog are case sensitive; therefore, upper and lower case letters uniquely distinguish labels. VHDL is case insensitive; therefore, upper and lower case letters within labels offer no further information. Figure 7 is the label conversion routine written in LEX. The LEX conversion routine simply examines the cif file for occurrences of a "94 label" pattern. The "94 label" pattern distinguishes a label record in the cif file from other records that describe the VLSI design geometry. The whole label is then converted to upper case except for Vdd. The LEX conversion routine is executed prior to circuit extraction by Mextra.

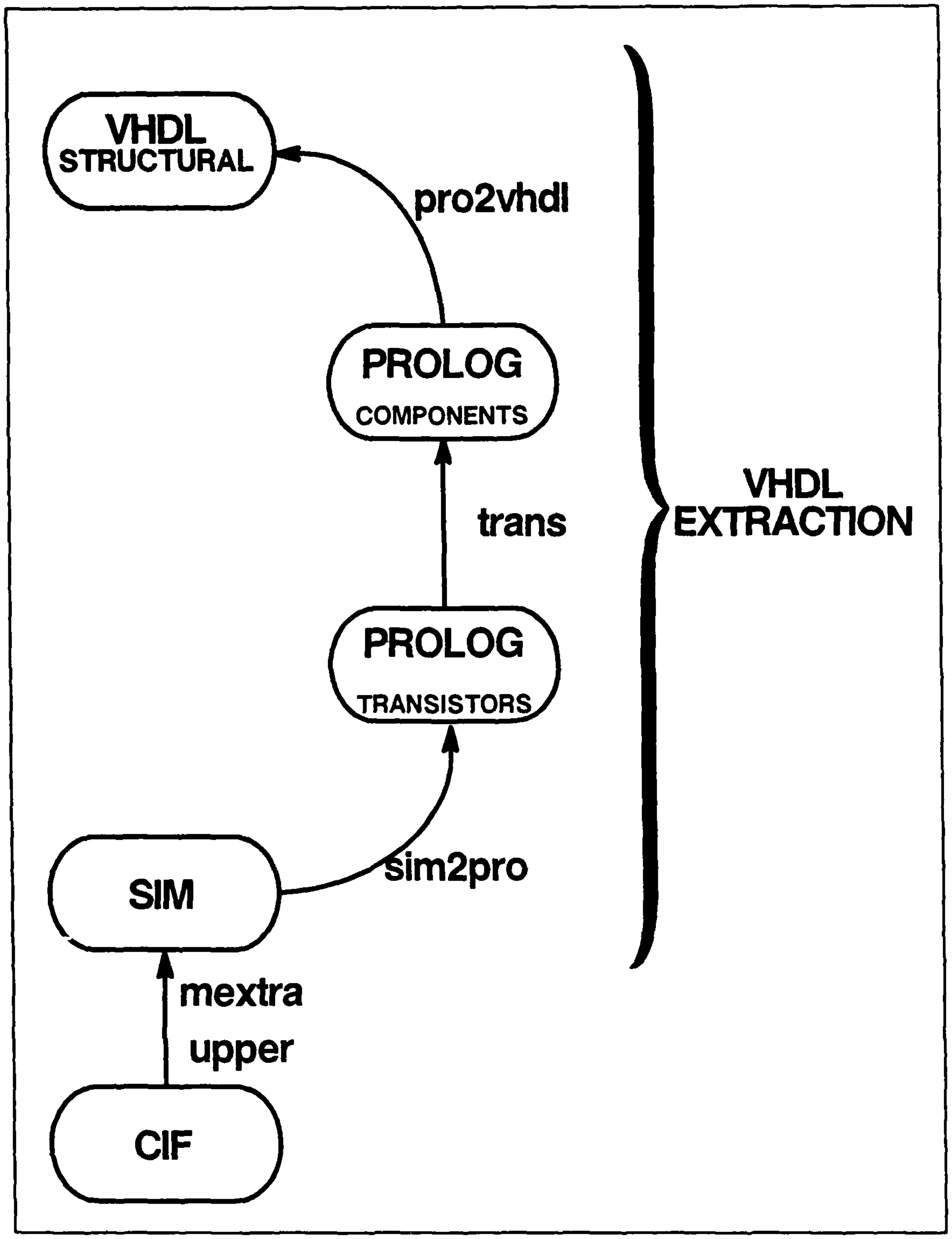


Figure 6. Sim to VHDL Extraction Process.

Conversion of Switch-Level Representation to Prolog

A software tool was necessary to convert the switch-level representation, sim, to Prolog clause form. The software tool, called sim2pro, could also be adapted to recognize other netlist forms of representation for transistors. The Prolog clause form was chosen as an intermediate representation that could be operated on easily within the Prolog environment.

From the switch-level representation for esim, called sim, a routine written in C performs a line-by-line translation from the sim format to a Prolog representation as in Figure 8. In the sim file, the "p" represents a p-type MOS transistor and the "e" represents an n-type or enhancement MOS transistor. The"d" for depletion mode and "f" for funny transistors are not implemented since they are not normally encountered within the VLSI designs at AFIT. The "n" is added as the first character of a label since a label from a sim file may begin with an uppercase character. The vdd and gnd labels in the Prolog clause form are not preceded by an "n" in order to help identify possible labeling problems that may have occurred in the layout. The Vdd and GND labels are assumed to represent power and ground within the chip design. Therefore, these labels are kept as the only reference points for the pattern matching performed later.

```
%START AA BB
int i;
%%
/*
/*  The following lines pass the Vdd! and GND! labels
/**/
^94[ ]Vdd![ ]([0-9]|\-)[^\n]*/\n            ECHO;
^94[ ]GND![ ]([0-9]|\-)[^\n]*/\n            ECHO;
/*
/*  The following block converts all lower case letters
/*  to upper case letters.
/*
^94[ ]              {
                    ECHO;
                    BEGIN AA;
                    }
<AA>[^ ]+           {
                    for (i=0; i<yyleng; i++)
                    if (yytext[i] >= 'a' && yytext[i]<='z')
                    for (i=0; i<yyleng; i++)
                       printf("%c",(yytext[i]+'A'-'a'));
                    else
                       printf("%c",yytext[i]);
                    BEGIN BB;
                    }
<BB>[^\n]*/\n       {
                    ECHO;
                    BEGIN 0;
                    }
```

Figure 7. LEX Routine to Convert Labels.

```
                sim          =>          pro
type   gate   drain  source     type   gate  drain  source

p      labelx  labely labelz     p(nlabelx,nlabely,nlabelz).
e      labelx  labely labelz     n(nlabelx,nlabely,nlabelz).


.sim =>      p 128 OZ_pq1 Vdd 300 21596 -26700 -24000

                          to

.pro =>              p(n128,nOZ_pq1,vdd).
```

Figure 8. Sim-to-pro Translation.

Component Extraction

With the transistor information in the Prolog format, the switch-level representation of the circuit can be read into Prolog as a database. The extraction process consists of a collection of Prolog clauses that contain pattern matching rules for components. The pattern matching consists of matching the correct transistor type, then matching the labels representing the gate, source, and drain. Figure 9 is an example of how the p and n-type MOS transistors are treated. The drain and source are interchangeable in MOS. Therefore, it is necessary to define rules in Figure 9 that accommodate this property.

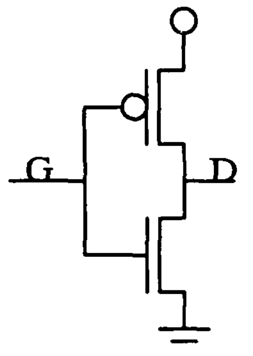
```
ptrans(G,D,S) :-          ntrans(G,D,S) :-
  p(G,D,S).                 n(G,D,S).
ptrans(G,D,S) :-          ntrans(G,D,S) :-
  p(G,S,D).                 n(G,S,D).
```

Figure 9. Prolog Representation of p and n-type MOS Transistors.

Some basic heuristics were developed to minimize the depth-first search inherent to Prolog execution. Basically, the areas of greatest concern were the database input and elimination of redundant transistors, extraction of the most common low-level components, and extraction of the higher level components through identification of basic signature subcomponents. Every attempt was made to exhaust an occurrence of a component in a Prolog clause as soon as possible to minimize nonessential searches. Furthermore, goals were aligned in the Prolog clauses such that discovery of the preceding subcomponent would lead to recognition of the next subcomponent from the database. An example can be constructed to illuminate this point.

Consider a description for a register cell constructed from two D flip-flops, a multiplexer, and two inverters. On a chip that performs arithmetic operations, register cells usually make up a small percentage of the collection of the chip components. D flip-flops then occur in low numbers, whereas, inverters and multiplexers exist in relatively large numbers. The number of D flips-flips can be considered to be less than the number of inverters or multiplexers. If in the process of extracting a register cell the inverters are examined first, then every inverter would be tested for its existence within a register cell. Testing D flip–flops first would incur fewer number of queries on the database than testing inverters. Once a D flip–flop is found, its output node may be used to help find the following D flip–flop and other surrounding inverters and multiplexer. In this manner, combinations involving two D flip–flops are exhausted rather than searching all possible combinations of an inverter followed by an interconnecting D flip–flop. Once two interconnecting D flip–flops are found, then the rest of the interconnecting components may be found by a direct lookup on the component database.

The next step was to establish an extraction methodology. Some components are constructed from lower–level components which are in turn constructed from components at still lower levels. For efficiency, a preferred extraction mechanism involves extracting the lowest level components first, so as to minimize the overall number of queries on the database. The first phase of extraction generates queries almost entirely on the transistor database. Thus, the first phase consists of extracting inverters, transmission gates (tgates), clocked inverters, ROM cells, and RAM cells. These low level elements cannot be decomposed into lower level representations other than transistors. Figure 10 is an example of a CMOS inverter and the Prolog code used to identify it.



```
inv :-
   ptrans(G,D,vdd),
   ntrans(G,gnd,D),
   remove_p(G,D,vdd),
   remove_n(G,gnd,D),
   asserta(inv(G,D,1)),
   fail.
inv.
```

Figure 10. CMOS Inverter Circuit with Prolog Description.

The remove_p and remove_n procedures remove n and p transistors from the database irrespective of the orientation of the drain and source. The 1 notation in the asserta(inv(G,D,1)) statement is used to identify the layout style of the inverter. The 1 notation serves to provide additional information about the design if it is necessary to break the component out into its transistor representation. Figure 11 demonstrates various circuit layouts considered for static CMOS design. Figure 12 also provides some circuit layouts for p or nMOS design. Figure 13 displays circuit configurations for transmission gate logic. Though not depicted in a figure, another form of layout style commonly generated makes use of precharged logic. Table 4 is a synopsis of the types of common basic circuit elements that maybe encountered within a layout style. The reason some basic circuit elements are not listed under certain layout styles is that they may be either derived from the basic circuit elements listed in that layout style or they are not usually constructed in that particular layout style.

The second phase of extraction generates the next levei of components, consisting entirely of transistors and components from the first phase. These components are various flip-flops, logical gates, multiplexers, half adders, full adders, and subtracters. These components usually make up a complete VLSI design cell.

Table 4. Logical Elements Categorized by Layout Style.

Static CMOS	n/pMOS	Tgate MOS	Precharged
NAND	NAND		NAND
NOR	NOR		NOR
Inverter	Inverter		"Logical Networks"
XNOR	XNOR	XNOR	
XOR	XOR	XOR	
"Logical Networks"		OR	
		AND	
		MUX	

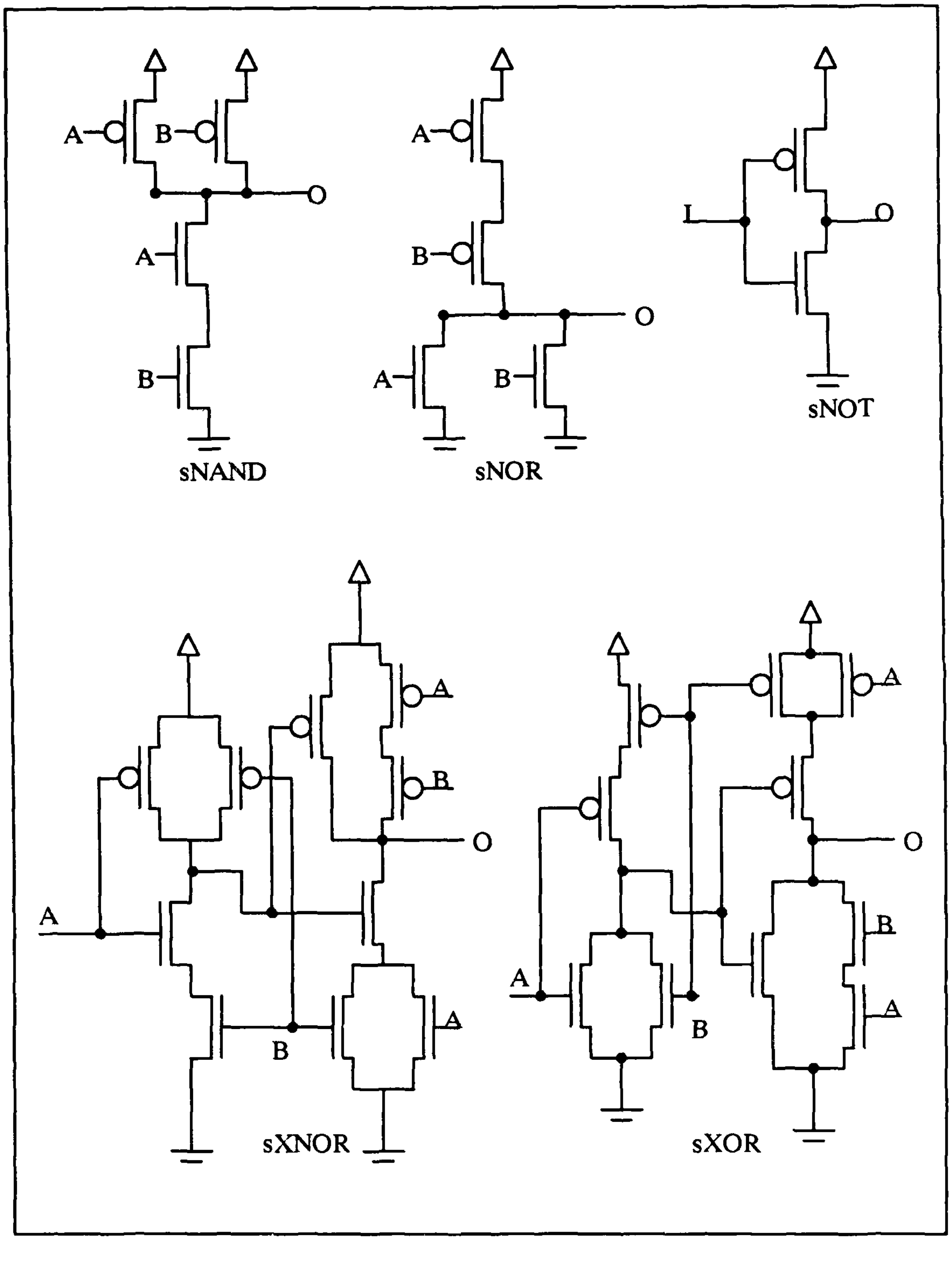


Figure 11. Typical Static CMOS Circuits.

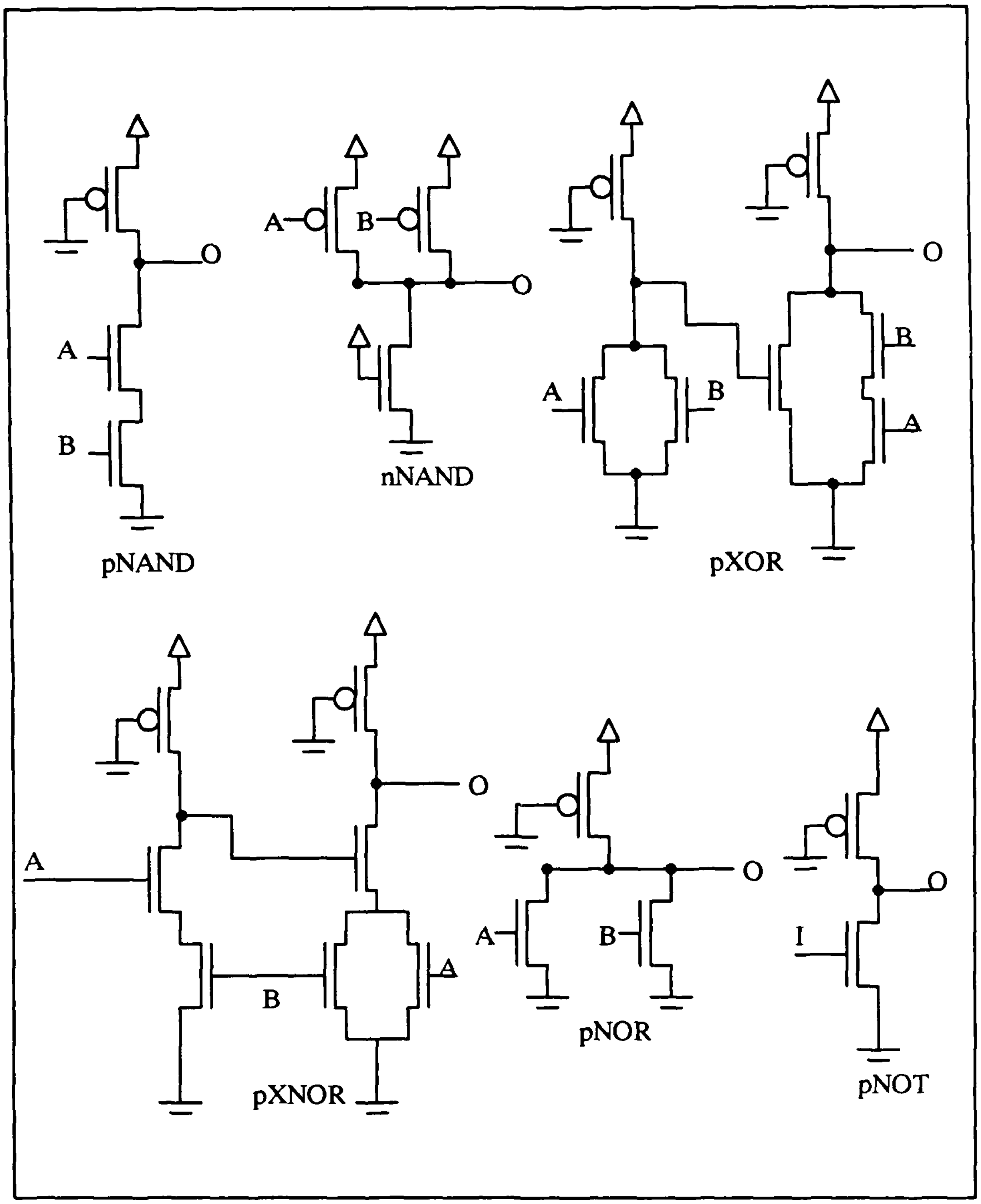


Figure 12. Typical p and nMOS Circuits.

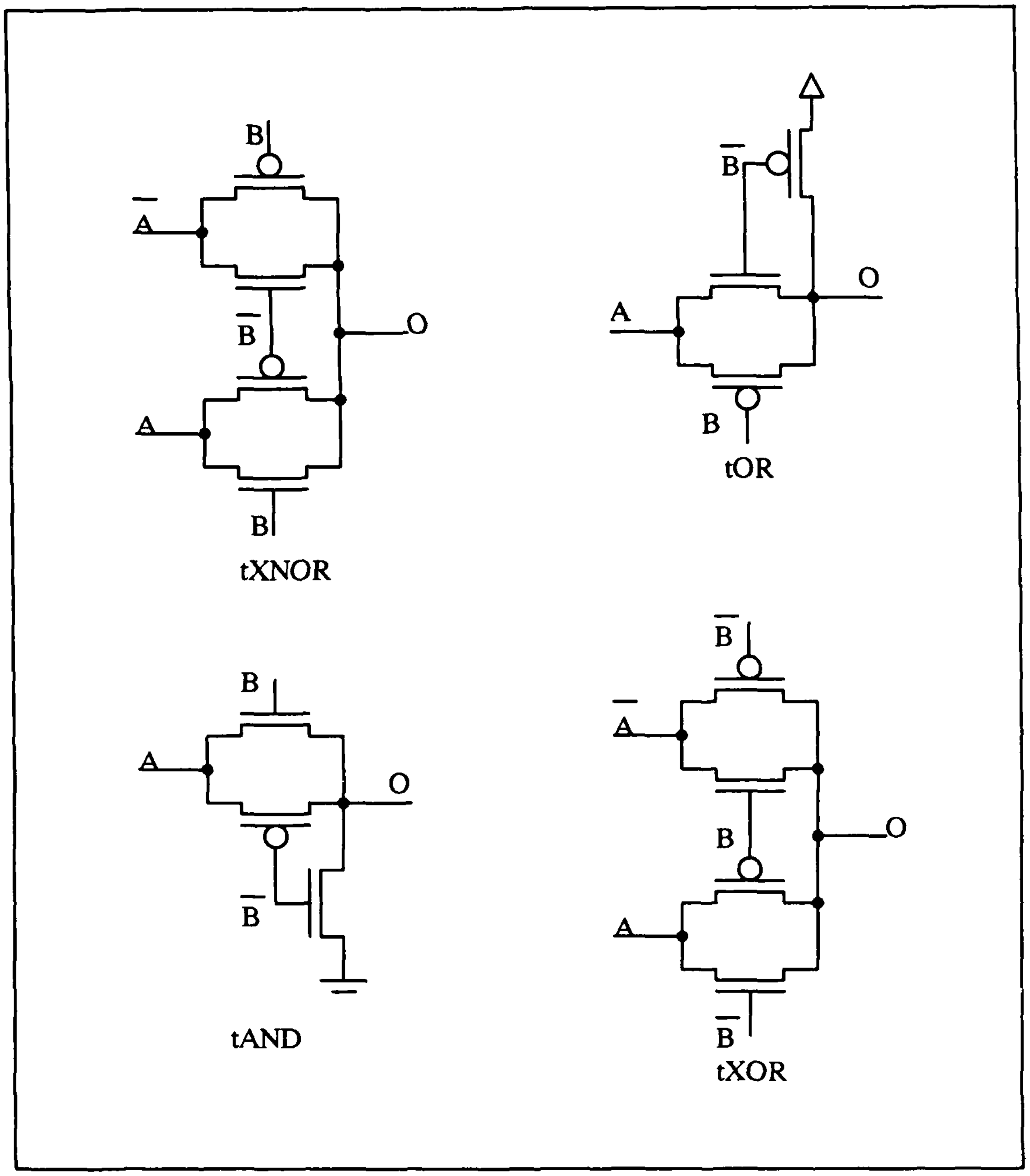


Figure 13. Typical Transmission Gate MOS Circuits.

Since logical gates may handle an "unlimited" number of possible inputs, the logic gate component construction uses a list for the input. Figure 14 is the Prolog code used to define a NAND gate in static CMOS. The main clause, 'nand', begins the description process for a NAND gate. It searches for a p-type transistor and n-type transistor that provide a basis for a NAND gate. Once an input has been found, 'nand' calls upon another Prolog Horn clause, 'more_nand', to collect corresponding n-type and p-type transistors. When only one set of p-type and n-type transistors is left, the second 'more_nand' Prolog clause will becalled. From this point, the calls to the 'more_nand' Prolog clauses are "unwound," building the list of inputs to the NAND gate. Notice, too, that the process of identifying transistors begins with an n-type transistor connected to GND and a neighboring n-type transistor and a p-type transistor connected to Vdd and any output. The process continues until a p-type and n-type transistor are discovered sharing a common output.

The third phase of the logic extraction process consists of identifying the macrocells using previously extracted components. The components created from this phase may be registers, memories, decoders, n-bit adders, n-bit subtracters, n-bit multipliers, and counters. A further extraction process may continue with the identification of macrocells.

```
nand :-
   ntrans(A,gnd,X),
   ptrans(A,vdd,O),
   not(gnd=O),
   not(vdd=O),
   not(X=O),
   more_nand(L,X,O,gnd),
   remove_p(A,vdd,O),
   remove_n(A,gnd,X),
   asserta(nand([A|L],O,1)),
   fail.
nand.

more_nand([A|L],X,O,P) :-
   ntrans(A,X,Y),
   ptrans(A,vdd,O),
   not(X=Y),not(Y=P),
   more_nand(L,Y,O,X),
   remove_n(A,X,Y),
   remove_p(A,vdd,O).

more_nand([A],X,O,_) :-
   ntrans(A,X,O),
   ptrans(A,vdd,O),
   not(X=O),
   remove_n(A,X,O),
   remove_p(A,vdd,O).
```

Figure 14. Static CMOS NAND in Prolog.

Ten-Level Multiple-Valued Logic Algebra

The proposed 10-level multiple-valued logic system accounts for a logically undefined state, three signal-strength levels, and three logical levels. A methodology for resolving capacitive charging is also presented. The following provides a general description for the ten-level multiple-valued logic system.

Logically Undefined. The BB or Base state is designated on nodes that have not been electronically affected by the surrounding circuitry within the simulation. For MOS transistors, a change that occurs on the drain will effect a change on the source. The same is also true for changes that occur on the source inducing changes on the drain. However, changes to the signal value on the gate of a transistor will not force the drain and source to change if both of the present values are in the BB state. In this case, the gate is considered to be electronically isolated from the source and drain.

Three Logical Strength Levels. The three strength levels used are Driven, D, Weak, W, and Charged, C. Driven nodes are separated from power or ground by three or less standard channel length transistors for the given technology (e.g. using a scaled design rule of lambda = 1.5 microns, standard channel length equals 3 microns). The Weak node is characterized by separation from power or ground through a resistive pullup or pulldown transistor. The Charged state is characterized by a node separated from power and ground due to "off" transistors. The strength relationship is $D > W > C$.

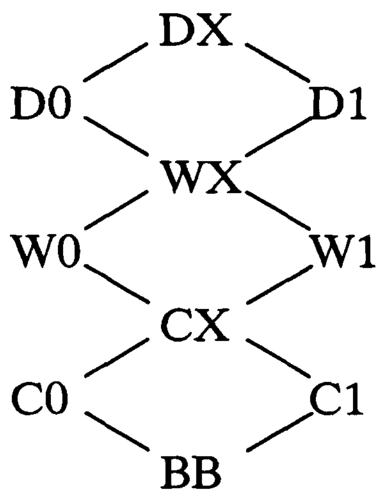
Three Logical Values. The three logical values are 1, 0, and X. The 1 state designates the existence of a voltage level very close to Vdd on a node. The 0 state indicates the existence of ground on a node. Finally, the X state designates the simultaneous existence of 1 and 0 of the same strength level on a node except as indicated under the definition of the capacitive level.

Capacitive Level. The capacitive level on a node may be represented in picofarads. Its use determines the value for interconnected capacitively charged nodes. The new charged values are calculated from the relation of the capacitance of one node compared to the capacitance of the surrounding nodes. The sum of the nodes with a capacitive value equal to or exceeding the sum of the other values will determine the logical level for the node under consideration. When neither a 1 nor a 0 condition can be determined from this method, the X state will be assigned to the node. The summation resolution method used in the case of capacitive level nodes is necessary to resolve out the pessimistic view of X states throughout a circuit due to charge sharing designs.

Algebra.

Let $P = < M, \vee, \wedge, BB >$ denote a partially ordered set (poset) with carrier M, meet and join relations $\wedge$ and $\vee$, and distinguished element BB.

Let M = {BB, C1, C0, CX, W1, W0, WX, D1, D0, DX} be the carrier and defined by strengths is exhibited by the Hasse diagram shown in Figure 15. The properties of idempotence, commutativity, associativity hold for the poset, M, exhibited in Figure 15 (Ullman, 1984:412). Only the join operation will be considered for the application of the multiple-valued logic system.



$< M, \vee, BB >$

for $a, b, c \in M$

$a \vee b = b \vee a$

if $c \geq a \vee b$ then

$(a \vee b) \vee c = a \vee (b \vee c) = c$

Figure 15. Strength Lattice for M.

However, for capacitive charges, this algebra exhibits an overly pessimistic view of actual MOS circuit behavior.

Let $S = \{BB, C1, C0, CX\}$ and $S \subset M$.

However, S is not defined by P. Consider that when the values on all drivers are contained within S a new relation is defined for capacitive values as in Figure 16. The $C\omega$ notation is a capacitive weighting method where ω defines the weight in units of Farads. The resolution for capacitive charge is valid iff the values under consideration are contained within S; otherwise the resolution for M must hold.

$$c = C1 \text{ iff } \left(\sum_{k=1}^{p} C\omega_k 1\right) \geq 2\left[\left(\sum_{i=1}^{n} C\omega_i X\right) + \left(\sum_{j=1}^{m} C\omega_j 0\right)\right]$$

OR

$$c = C0 \text{ iff } \left(\sum_{j=1}^{m} C\omega_j 0\right) \geq 2\left[\left(\sum_{i=1}^{n} C\omega_i X\right) + \left(\sum_{k=1}^{p} C\omega_k 1\right)\right]$$

otherwise c = CX

Figure 16. Capacitive Relation for S.

Prolog to VHDL

The problem of generating VHDL from Prolog was considered as a less important goal than a ten-level multiple-valued logic system. Since the problem was a straight mapping of Prolog components onto VHDL component instantiations, a simple solution was incorporated into the design of this software routine. Within the implementation chapter of this thesis, this consideration will become apparent.

The approach to this problem was to initially test for the existence of components within the Prolog database and output a VHDL design entity that represented the Prolog component. Once all of the necessary design entities were created, the entity for the test bench and architecture was created followed by the signal declaration statements. The next step involved the creation of the component specifications, the begin statement, and test bench control process.

Afterwards, all of the VHDL components were instantiated from components in the Prolog database. Finally, the end statement for the architecture was placed at the end of the generated VHDL specification.

V. IMPLEMENTATION

Introduction

This chapter will cover the implementation of the ten-level multiple-valued logic system in VHDL and the logic extraction system in Prolog. The areas covered under the ten-level multiple-valued logic system include the MOS_logic_package, truth tables for p-type and n-type MOS transistors, a description of the transistor design entities, and an example of the node resolution algorithm. The items covered in the software implementation of the logic extraction system include a description of the heuristics in the extraction routine, establishment of signature subcomponents within higher level components, and the conversion from Prolog clause form to VHDL.

Ten-Level Multiple-Valued Logic

The MOS_logic_package package establishes the type definitions used throughout the VHDL design. Figure 17 is a listing of the types and Figure 18 is a listing of the functions declared in the MOS_logic_package. The strength lattice is established as a construction, MOS_logic, of two enumerated types, strength and value. The MOS_node record also contains the capacitive value for the node.

```
type trans_type is ('P','N','U','D');

--  For trans_type the following mean:
--
--  P : Optimally sized p-type enhancement MOS.
--  N : Optimally sized n-type enhancement MOS.
--  U : Resistively sized p-type enhancement MOS.
--  D : Resistively sized n-type enhancement MOS.

type strength is ('B', 'C', 'W', 'D');

type value    is ('B', '0', '1', 'X');

type capacitance is range 0 to 100000;

type MOS_logic is record
   S : strength;
   V : value;
   end record;

type MOS_node_record is record
   L : MOS_logic;
   C : capacitance;
   end record;

type MOS_node_and_time is record
   N : MOS_node_record;
   T : time;
   end record;

type MOS_node_array is array (Natural range <>) of
   MOS_node_and_time;

type mos_node_resolution_array is array (integer range <>)
   of MOS_node_record;

function mos_node_resolution (input : mos_node_resolution_array)
   return mos_node_record;

subtype mos_node is mos_node_resolution mos_node_record;
```

Figure 17. Basic Package Specification for MOS.

```
function snand (A,B : MOS_node_record)
    return MOS_node_record;

function snor  (A,B : MOS_node_record)
    return MOS_node_record;

function snot  (A : MOS_node_record)
    return MOS_node_record;

function sxnor (A,B : MOS_node_record)
    return MOS_node_record;

function pnand (A,B : MOS_node_record)
    return MOS_node_record;

function nnand (A,B : MOS_node_record)
    return MOS_node_record;

function pnor  (A,B : MOS_node_record)
    return MOS_node_record;

function pnot  (A : MOS_node_record)
    return MOS_node_record;

function tmux  (A,B,S,Sbar : MOS_node_record)
    return MOS_node_record;

function dff   (A, PHI, PHI_bar, OUTput : MOS_node_record)
    return MOS_node_record;

function binary_to_multi (A : bit)
    return MOS_node_record;

function multi_to_binary (Sig : bit; A : MOS_node_record)
    return bit;
```

Figure 18. Specification of Supporting Functions for MOS.

This package contains functions for the predefined circuit representations from Figures 11, 12, and 13. The circuits are defined in terms of their transistor configurations. Once the transistor configuration is established, the expected outputs can then be specified in terms of the inputs provided.

Transistor Function Tables. Once the multiple-valued logic system has been defined, truth tables may be constructed for the n-type and p-type MOS transistors. Table 5 and Table 6 are the truth tables for the n-type and p-type MOS transistors.

NOTE: For Tables 5 and 6, the input and output nodes of the MOS transistor may be interchanged with the source and the drain depending on the application.

Table 5. Truth Table for n-type MOS Transistor.

Input	Gate	Output
BB	d	BB
C0	BB,DX,WX,CX	C0
C1	BB,DX,WX,CX	C1
CX	BB,DX,WX,CX	CX
D0,W0	BB,DX,WX,CX	W0
D1,W1	BB,DX,WX,CX	W1
DX,WX	BB,DX,WX,CX	WX
DX,D1,D0,WX,W1, W0,CX,C1,C0	D0,W0,C0 (Capacitive weight of 0.0)	CX
D1	D1,W1,C1	W1
W1	D1,W1,C1	W1
C1	D1,W1,C1	C1
D0	D1,W1,C1	D0
W0	D1,W1,C1	W0
C0	D1,W1,C1	C0

Table 6. Truth Table for p–type MOS Transistor.

Input	Gate	Output
BB	d	BB
C0	BB,DX,WX,CX	C0
C1	BB,DX,WX,CX	C1
CX	BB,DX,WX,CX	CX
D0,W0	BB,DX,WX,CX	W0
D1,W1	BB,DX,WX,CX	W1
DX,WX	BB,DX,WX,CX	WX
DX,D1,D0,WX,W1, W0,CX,C1,C0	D1,W1,C1 (Capacitive weight of 0.0)	CX
D1	D0,W0,C0	D1
W1	D0,W0,C0	W1
C1	D0,W0,C0	C1
D0	D0,W0,C0	W0
W0	D0,W0,C0	W0
C0	D0,W0,C0	C0

The truth tables serve to provide a first analysis of the result of a logic operation through a transistor. Figure 19 is a VHDL design entity used to calculate the logical result on an output from a transistor. The final resolved value on a node depends upon resolving the values generated by all surrounding transistors through a bus resolution function shown in Figure 20.

```
entity ptrans is
   port (Gate   : in mos_node;
         Drain  : inout mos_node;
         Source : inout mos_node);
   end;

architecture ptrans of ptrans is
   begin
   process(Gate, Drain, Source)
      variable Drain_temp : mos_node_record;
      variable Source_temp : mos_node_record;
      begin
      If (Gate.L.V = '0') then
         If((Drain.L.S='D') and (Drain.L.V='0')) then
            Source_temp.L.S := 'W';
            Source_temp.L.V := '0';
         else
            Source_temp := Drain;
            end if;
         If((Source.L.S='D') and (Source.L.V='0')) then
            Drain_temp.L.S := 'W';
            Drain_temp.L.V := '0';
         else
            Drain_temp := Source;
         end if;
      elsif ((Gate.L.V = 'X')or(Gate.L.V = 'B')) then
         If (Source.L.S = 'D') then
            Drain_temp.L.S := 'W';
            Drain_temp.L.V := Source.L.V;
         else
            Drain_temp := Source;
            end if;
         If (Drain.L.S = 'D') then
            Source_temp.L.S := 'W';
            Source_temp.L.V := Drain.L.V;
         else
            Source_temp := Drain;
            end if;
      else
         Drain_temp.L.V := Drain.L.V;
         Drain_temp.L.S := 'C';
         Source_temp.L.V := Source.L.V;
         Source_temp.L.S := 'C';
         end if;
      Drain <= Drain_temp after 1 ns;
      Source <= Source_temp after 1ns;
      end process;
   end;
```

Figure 19. P Transistor Design Entity.

```
function mos_node_resolution
        (input : mos_node_resolution_array)
             return mos_node_record is

  variable output, temp : mos_node_record;

  begin
  output.L.S := 'B';
  output.L.V := 'B';
  for i in input'range loop
     temp := input(i);
     If (temp.L.S > output.L.S) then
        output := temp;
     elsif ((temp.L.S = output.L.S) and
           (temp.L.V /= output.L.V)) then
        output.L.V := 'X';
        end if;
     end loop;
  return(output);
  end mos_node_resolution;
```

Figure 20. Node Bus Resolution Function.

Node Resolution Algorithm. Determination of node values is based upon a sequence of steps that closely maps to the signal value resolution within the VHDL simulator. Every node has at least one driver. A transistor supplies two drivers, one for the drain and one for the source.

Step 1: Initialize all nodes to BB.

Step 2: Evaluate all stimuli drivers and post future transactions.

Step 3: If no transactions are due, halt simulation.

Step 4: Compare transactions due at current time to values on appropriate nodes. If different, change the node value to the new transaction value. Else, discard the transaction.

Step 5: Reevaluate drivers changed by node changes in step 3.

Step 6: Resolve future values on nodes modified by drivers in step 5.

Step 7: Post new transactions from resolution in step 6 and return to step 3.

Node Resolution Example. Consider the simple circuit in Figure 21. Assume the logical values for all nodes in Table 7 at time T and the list of drivers in Table 8.

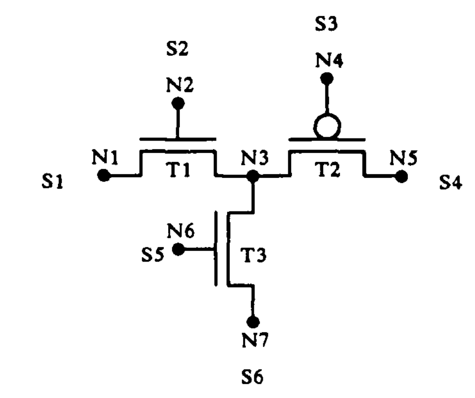


Figure 21. Example of Node Resolution

Table 7. Values for Circuit in Figure 20 at times T to T + 4.

Node	Value and Time				
	T	T + 1	T + 2	T + 3	T + 4
N1	BB	BB	D1	D1	D1
N2	BB	D0	D0	D0	D0
N3	BB	BB	BB	CX	CX
N4	BB	D0	D0	D0	D0
N5	BB	BB	BB	BB	CX
N6	BB	D0	D0	D0	D0
N7	BB	BB	BB	BB	CX

Table 8. Table of Drivers.

Node	Driver	Node	Driver
N1	S1.N1	N4	S3.N4
N1	T1.N1	N5	S4.N5
N2	S2.N2	N5	T2.N5
N3	T1.N3	N6	S5.N6
N3	T2.N3	N7	S6.N7
N3	T3.N3	N7	T3.N7

S1, S2, S3, S4, S5, and S6 are all external stimuli drivers. Assume D0 is assigned to nodes N2 and N6 via the drivers S2.N2 and S3.N4 respectively. No change will occur to any other nodes since they are electrically isolated from the circuit. The value on node N4 then changes to D0. Again, no further changes will occur. Table 7 reflects the values at T + 1.

During the T + 2 iteration, N1 is set to a new value of D1 from driver S1. From the truth table for an n-type MOS transistor the output value on transistor T1 should be a CX with a weighted capacitance of 0.0 pF. The value on N3 with respect to transistor T1 may be designated as T1.N3 as may the value on N3 with respect to T2 may be designated T2.N3 and finally T3.N3. The final value on N3 may be resolved from the following equation as

$$N3 = T1.N3 \lor T2.N3 \lor T3.N3 \tag{1}$$

$$N3 = CX \lor BB \lor BB \tag{2}$$

The future value to be assigned to N3 at T + 3 is then CX. Table 7 lists the values for all nodes at T + 2.

At T + 3, N3 is assigned the new value of CX. At this point new future values for the connected nodes must be determined. From the truth table for n-type MOS transistors the new possible future value for N1 is CX. This value will be stored in a driver called T1.N1 *resulting from* N3. N1 is the home node for the following evaluation.

$$N1 = S1.N1 \lor T1.N1 \tag{3}$$

From equation 3, the future value of N1 should remain D1. For the evaluation on node N5, the value for the driver, T2.N5, is evaluated from the truth table for p-type MOS transistors. The result is CX for T2.N5. The driver, S4.N5, has a value of BB. The resolved future value for N5 then becomes CX. For node N7, the driver T3.N7 obtains a new value of CX. The driver S6.N7 is BB. The resulting future value for N7 is then CX. Table 7 contains the node values at T + 3.

At T + 4, the resolved value for N1 does not cause a change and is discarded. The new values for N5 and N7 are assigned. Reevaluation of the drivers T2.N3 and T3.N3 is performed with a new future value of CX on each. The three drivers T1.N3, T2.N3, and T3.N3 are then resolved to obtain a new future value assignment for N3 of CX at T + 5. Table 7 is a list of the node values at T + 4.

At T + 5, N3 does not change and no further evaluations of driver values is performed. The simulation would terminate at this point provided no further stimuli were provided on S1, S2, S3, S4, S5, or S6.

Software Implementation

This section outlines the implementation of the logic extraction process shown in Figure 6. The order of the discussion follows the order of tool execution shown in Figure 6 in order to maintain continuity in the discussion. Therefore, the tools upper, sim2pro, trans, and pro2vhdl will be elaborated in that order. The pro2vhdl tool will be described in the most detail since generation of simulatable VHDL is considered the most original portion of STOVE_P.

Labeling. In addition to case sensitivity, other labeling problems were recognized. In some designs, the apostrophe is used to distinguish the inverse of a signal as part of a designer's labeling convention. Some designers use hyphens as well as the pound sign and most any spurious character available. VHDL's character set for labeling conventions (IEEE, 1987: A-5 to A-6) is limited strictly to

```
identifier ::= letter {[ underline ] letter_or_digit }
letter_or_digit ::= letter | digit
letter ::= upper_case_letter | lower_case_letter
```

Within some VLSI designs it is common to also find digits as the first character of the label. Though these problems exist in the labeling conventions used throughout the designs in the AFIT CAD Environment, the implementation of upper only attacked the major problem of upper and lower case letter normalization. So that mextra could provide the proper numbering suffix for different nodes with similar labels, the upper routine was designed to operate on the cif file prior to the use of mextra.

Sim2pro. The sim2pro routine performed the conversion of the transistor netlist format from a sim file into Prolog clause form. Figure 8 demonstrates the field by field conversion involved. The language chosen to perform this task was C. The choice of C was due to the relative simplicity of the problem and the efficiency of using C in a Unix environment. As such, this routine was more suited to execution on one of the systems normally used for layout of VLSI designs. Appendix C is a listing of sim2pro. If necessary, sim2pro may be modified to accommodate translation from another transistor netlist representation.

Trans. Once the sim file was converted to Prolog clause form the trans routine could be executed to perform the extraction of components. This routine marked the point where a faster CPU was called upon, because of the CPU and memory intensive nature of the Prolog solution. A VAX 8800, running VMS, was the host system. Quintus Prolog was employed in the extraction process. An earlier developmental version of STOVE_P was run on a VAX 11/785 running Unix and CProlog.

The Prolog code produced for the extraction portion of STOVE_P is largely a template-based system as discussed earlier. The order of execution of the templates is crucial, since some groups of components contain some transistor configurations similar to those in smaller designs. A high impedance inverter is one example of a component that also appears as an inverter followed by a transmission gate. Even though a high impedance inverter and an inverter transmission gate combination are similar, the node between the inverter and transmission gate is typically part of a feedback loop in a D flip–flop. Extracting high impedance inverters first eliminates the node typically used for the feedback loop in a D flip–flop.

Several heuristics are incorporated into the execution of the Prolog code to help improve execution efficiency. Initially, the Prolog clause form of a transistor netlist was read in completely before eliminating duplicates. Duplicate transistors were distinguished by two or more transistors of the same type having a common gate, drain, and source. The connectivity between the drain and source could be either drain–to–drain and source–to–source or drain–to–source and source–to–drain. The subsequent elimination of duplicates required a large amount of stack space and procedure calls, further requiring additional CPU time and page faults. The input routine for reading the Prolog clause form of the transistor netlist was therefore changed to query the database for the existence of a duplicate transistor before adding it to the data base. This procedure now uses very little stack space and makes fewer procedure calls.

A second and highly significant heuristic employed in the extraction process involved forcing failure on an extraction when failure was imminent. Since a search for any Prolog routine is largely depth first, a great deal of CPU time and memory can be spent attempting to resolve every possible combination, whether it is successful or not. Therefore, critical subcomponents of larger components are identified as signatures for the larger components. An example of a signature for a register cell is a D flip-flop. Another example of a signature for a large multiple-bit adder macrocell is a full adder.

Once a signature subcomponent is identified for a larger component, the signature component is the first component to be examined in the components database. Figure 22 is an example of a Prolog Horn clause for identifying a particular register cell. The D flip-flops are identified by the dff prefix. The fields within dff specify clock bar, clock, input, and output. For the buffers, buffer identifies the component, with fields for input and output respectively. For the 2-to-1 multiplexers, mux identifies the component followed by fields for the select, select bar, input 1, input 2, and the output. For the inverter, inv identifies the component, followed by the input and output fields respectively. The information conveyed by the Prolog Horn clause specifies the existence of a D flip-flop first. If one does not exist, then failure will occur, eliminating any further queries on the component database. If one does exist, it must have its output connected to the input of another D flip-flop as asserted by the second dff component. Once again, if this condition does not occur, then failure of the routine will occur. An assumption can be made at this point that the likelihood of failure is slim if both D flip-flops exist and if the VLSI layout design consists mostly of register cells of the type specified. That being the case, the buffer, multiplexer, and inverter should follow very quickly since information from the instantiations of the D flip-flops may be used to assist in their lookup.

```
in_2reg_cell :-
   dff(NLD_PQ1_bar,NLD_PQ1,NIN,N27,1),
   dff(Nphi2bar,NPQ2,N27,N23,1),
   buffer(N23,NOUT,1),
   mux(N1,Nsbar,N72,NA,NIN,3),
   inv(N1,Nsbar,1),
   retract(dff(Nphi2bar,NPQ2,N27,N23,1)),
   retract(dff(NLD_PQ1_bar,NLD_PQ1,NIN,N27,1)),
   retract(buffer(N23,NOUT,1)),
   retract(mux(N1,Nsbar,N72,NA,NIN,3)),
   retract(inv(N1,Nsbar,1)),
   asserta(in_2reg_cell(N1,N72,NA,NPQ2,Nphi2bar,
           NLD_PQ1,NLD_PQ1_bar,NOUT,1)),
   fail.
in_2reg_cell.
```

Figure 22. Prolog Definition for a Register Cell.

Once all of the components specified have been found, they are then removed from the database and a register cell called in_2reg_cell is inserted. The last statement, fail, of the Prolog Horn clause is used to force the Prolog Horn clause to backup and extract more components until all possible solutions have been attempted. One further heuristic that may be applied in this case is to first check for signature components that rarely occur in the components database. In this manner, the number of initial database queries is limited by the rarely-occurring components.

Pro2vhdl. The pro2vhdl routine was written in Prolog. The reason for the choice of Prolog was for the ease of expression of the software solution. Figure 23 is the main procedure that controls the execution of the Prolog code. The main procedure is written as a Horn clause. 'go1' reads in the SIF of the file. The 'tell' procedure is defined in Clocksin and Mellish for specification of the output file. 'pass1' generates the design entities and architectures for the design components that will be used within the test bench. 'pass1' also generates the test bench entity, architecture statement, component specification, test bench control process, and signal declaration statements.

```
go :-
    go1,
    tell('outfile.vhd'),
    pass1,
    pass2,
    told,
    halt.
```

Figure 23. Main Procedure for Pro2vhdl.

Figure 24 shows the basic composition of the 'pass1' and 'pass2' procedures. 'pass1' and 'pass2' are also Horn clauses. Within 'pass1', 'gen_basic_comp' generates the design entities; 'gen_header_1' produces the entity and architecture statement for the test bench; 'gen_sig' creates the signal assignment statements; 'gen_header_2a' constructs the component specifications for those design entities referenced in the testbench; and 'gen_header_3' generates the process statement to control the simulation. 'pass2' uses 'gen_body' to create the component_instantiation_statement (IEEE 1987: A-5). 'gen_tail' produces any further VHDL statements necessary to complete the architecture of the test bench.

```
pass1 :-
    gen_basic_comp,
    gen_header_1,
    gen_sig,
    gen_header_2a,
    gen_header_3.

pass2 :-
    gen_body,
    gen_tail.
```

Figure 24. 'pass1' and 'pass2' Procedures for Pro2vhdl.

Specifying the generation of VHDL code was performed in a straightforward manner. Figure 25 is an example of a Prolog Horn clause that produces a complete design entity description for a transmission gate. For every design entity that references the transmission gate, a test for the existence of an adder, xnor, xor, as well as the transmission gate is performed prior to the output of the design entity. If none of these components exist in the Prolog database, then the design entity will not be produced in the output file. An example of a Prolog Horn clause used in the production of a configuration specification for a high impedance inverter is shown in Figure 26. The configuration specification must exist in the architecture declarative section of the test design entity in order for instantiations of high impedance inverters in the architecture body to analyze correctly.

```
gen_tgate :-
   (adder(_,_,_,_,_,3);xnor(_,_,_,_,_,3);
   xor(_,_,_,_,_,3);tgate(_,_,_,_)),
   write('use work.mos_logic_package.all;'),nl,
   write('entity tgate is'),nl,
   write('      port (signal p1 : in mos_node;'),nl,
   write('           signal p2 : in mos_node;'),nl,
   write('           signal g  : inout mos_node;'),nl,
   write('           signal d  : inout mos_node);'),nl,
   write('end tgate;'),nl,
   write('architecture tgate of tgate is'),nl,
   write('        component ptrans'),nl,
   write('               port (Gate   :in mos_node;'),nl,
   write('                    Drain  :inout mos_node;'),nl,
   write('                    Source :inout mos_node);'),nl,
   write('               end component;'),nl,
   write('      for all : ptrans'),nl,
   write('        use entity work.ptrans ( ptrans );'),nl,
   write('        component ntrans'),nl,
   write('               port (Gate   :in mos_node;'),nl,
   write('                    Drain  :inout mos_node;'),nl,
   write('                    Source :inout mos_node);'),nl,
   write('               end component;'),nl,
   write('      for all : ntrans'),nl,
   write('         use entity work.ntrans ( ntrans );'),nl,
   write('  begin'),nl,
   write('          PTRANS1:PTRANS port map( Gate =>P1,'),nl,
   write('                 Drain =>g,'),nl,
   write('                Source =>d);'),nl,
   write('          NTRANS1:NTRANS port map( Gate =>P2,'),nl,
   write('                 Drain =>g,'),nl,
   write('                Source =>d);'),nl,
   write('end tgate;'),nl.
gen_tgate.
```

Figure 25. A VHDL Design Entity for a Tgate in Pro2vhdl.

```
gen_head_clk_inv :-
    clk_inv(_,_,_,_,1),
    write('        component clk_inv'),nl,
    write('                port ( vdd  : inout mos_node;'),nl,
    write('                       gnd  : inout mos_node;'),nl,
    write('                       p1   : in mos_node;'),nl,
    write('                       p2   : in mos_node;'),nl,
    write('                       g    : in mos_node;'),nl,
    write('                       d    : inout mos_node);'),nl,
    write('                end component;'),nl,
    write('    for all : clk_inv use entity work.clk_inv (clk_inv);'),nl.
    gen_head_clk_inv.
```

Figure 26. A VHDL Component Specification in Pro2vhdl.

Signal statement generation in pro2vhdl is handled through the 'gen_sig' clause. Figure 27 shows a portion of the 'gen_sig' clause, as well as the 'add_signal', 'gen_sig_adder', and 'gen_sig_state' clauses. The operation of these clauses involves building a database of signals and generating the signal assignment statements for these signals. The database of signal names was developed so that duplicate signal names from components could be eliminated, since a signal name may not be declared multiple times in VHDL.

```
gen_sig :-
    !,add_signal(vdd),
    add_signal(gnd),
    gen_sig_inv,
    gen_sig_ptrans,
    gen_sig_adder,
    gen_sig_state.

add_signal(A) :-
    signal(A),!.

add_signal(A) :-
    asserta(signal(A)).

gen_sig_adder :-
    retract(adder(A,B,C,D,E,3)),
    add_signal(A),
    add_signal(B),
    add_signal(C),
    add_signal(D),
    add_signal(E),
    gen_sig_adder,
    asserta(adder(A,B,C,D,E,3)).
    gen_sig_adder.

gen_sig_state :-
    retract(signal(X)),
    write('          signal '),
    write(X),write(' : mos_node;'),nl,
    gen_sig_state.

gen_sig_state.
```

Figure 27. Signal Assignment Generation in Pro2vhdl.

In order for the VHDL simulation model to function within the confines of the test bench from the SIMULATOR STANDARD package, a process statement is generated that will start the simulation, initialize VDD and GND to a D1 and D0 respectively, and set a termination time for the simulator. Additional information may be placed in the process statement pertaining to test vectors and other initialization information that may be pertinent to the simulator of a particular model. Figure 28 is a Prolog Horn clause used for generation of the process statement.

```
gen_header_3 :-
   write('begin'),nl,
   write('process'),nl,
   write('     variable high_volt  : mos_node_record;'),nl,
   write('     variable low_volt   : mos_node_record;'),nl,
   write('     begin'),nl,
   write('     set_maximums(10000,100);'),nl,
   write('     tracing_on;'),nl,
   write('     high_volt.L.S := "D";'),nl,
   write('     high_volt.L.V := "1";'),nl,
   write('     low_volt.L.S  := "D";'),nl,
   write('     low_volt.L.V  := "0";'),nl,
   write('     vdd <= high_volt;'),nl,
   write('     gnd <= low_volt;'),nl,
   write('     wait;'),nl,
   write('     end process;'),nl.
```

Figure 28. Process Statement Generation in Pro2vhdl.

Once the process statement for initializing the simulation model has been generated, the instantiations of the components are generated within the body of the architecture. The 'gen_body' Prolog Horn clause performs two functions. The first function is to provide a numbering scheme that will differentiate each instantiation of a component from others of the same type. As each instantiation is produced, the number corresponding to its order of occurrence is appended to the instantiated component's name. The second purpose of the 'gen_sig' Prolog Horn clause is to insure the instantiation of the different types of component within the architecture body. A description of 'gen_body', 'adder_port', and 'gen_tail' are included in Figure 29. The 'adder_port' Prolog Horn clause also demonstrates how the signals within the Prolog component are matched to the correct position within the port map for the adder.

```
[illegible]_body :-
    X is 0,
    ptrans_port(X),
    ntrans_port(X),
    adder_port(X),
    in_2reg_cell_port(X).

adder_port(Y) :-
    X is Y + 1,
    retract(adder(A,B,C,D,E,3)),
    write('        adder'),write(X),
    write(':adder port map ( vdd     => vdd,'),nl,
    write('                     gnd     => gnd,'),nl,
    write('                     nA      =>'),write(A),write(','),nl,
    write('                     nB      =>'),write(B),write(','),nl,
    write('                     ncin    =>'),write(C),write(','),nl,
    write('                     nsum    =>'),write(D),write(','),nl,
    write('                     ncyout =>'),write(E),write(');'),nl,
    adder_port(X).
adder_port(_).

gen_tail :-
    write('end testor;'),nl.
```

Figure 29. Generation of the Remaining Architecture Body in Pro2vhdl.

Verification and Validation

Even though generation of a production quality software CAD tool was not a goal of this research, some form of verification and validation of the software was necessary in order to maintain the integrity of the proof of concept. Therefore, it was necessary to test the extraction methodology and the ten-level multiple-valued logic system for its validity. The subsequent discussion will first center on the correctness of the extraction methodology. The cases used to test the ten-level multiple valued logic system will then follow.

For the sake of arguing logic extraction, consider the construction of a transmission gate. A transmission gate consists of a p-type transistor and an n-type transistor that share a common drain and source. A predicate calculus method for representing this relationship may then consist of:

P(x,y,z) : p-type transistor with gate x, drain y, and source, z.

N(x,y,z) : n-type transistor with gate x, drain y, and source, z.

Tgate(w,x,y,z) : transmission gate with w on the p-type transistor gate, x on the n-type transistor gate, y as an input/output and z as an input/output.

In English

IF a p-type transistor has a gate, a, drain b, and source, c, AND an n-type transistor has a gate, d, drain b, and source, c, THEN the configuration is a transmission gate with a on the p-type transistor gate, d on the n-type transistor gate, b as an input/output, and c as an input/output.

Applying predicate calculus

$$[\ P(a,b,c) \land N(d,b,c)\ ] \Rightarrow Tgate(a,d,b,c)$$

where the $\land$ denotes logical conjunction and => denotes logical implication. Of course, the source and drain are interchangeable and further clauses could be constructed to support this assertion for the p-type and n-type transistors.

Let :- be logical implication in the opposite direction and modify the representations to conform to the labeling conventions of Prolog, then

```
tgate(A,B,C,D) :-
   ptrans(A,C,D),
   ntrans(B,C,D).
```

and

```
ptrans(A,B,C) :-
   p(A,B,C).
ptrans(A,B,C) :-
   p(A,C,B).

ntrans(A,B,C) :-
   n(A,B,C).
ntrans(A,B,C) :-
   n(A,C,B).
```

The Prolog horn clauses for ptrans and ntrans specify the interchangeability of the drain and source. The assertion for tgate is then true for any transistor netlist of a CMOS design.

Given that Prolog templates may be constructed in the same fashion as described above, then only designs with components described with these templates will be extracted to a level higher than the transistor level. In order to test the validity of the templates generated in Prolog, several designs were chosen from the AFIT CAD Environment for extraction. The three designs chosen were a clock generator, 31-bit register macrocell, and a 31-bit carry select adder of 116, 806, and 2288 transistors respectively. Three weeks were spent analyzing the design components and extracted output for correspondence. Furthermore, this time was also spent analyzing the VHDL output from the extraction process.

In order to analyze the VHDL simulation results of the ten-level multiple-valued logic system, the clock generator and various cells of the register macrocell and 31-bit carry-select adder were simulated extensively using behavioral descriptions and esim output (esim output could be produced for designs without feedback). Essentially, design units previously laid out and simulated through another tool, as well as knowledge of the behaviors of these units, were used to validate the results from using the ten-level multiple-valued logic.

One caveat that should be mentioned concerns the status of VHDL 1076-1987. At the time of publication of this thesis, Intermetrics had released their Version 1.0 of VHDL 1076-1987. Reliability problems still persisted in the simulator concerning the use of composite types and bus resolution functions on signals. The VHDL functions, design entities, and test bench were originally tested in VHDL V7.2 using the Intermetrics Version 3.4 VHDL V7.2 environment. All of the functions, design entities, and test bench were translated to VHDL 1076-1987 and have thus far been used by the Air Force Wright Aeronautical Laboratory to identify bugs in the Intermetrics VHDL environment. Specifically, problems with composite types, bus resolution functions, signal labels, and packages are areas still under software debugging in the new VHDL 1076-1987 tool environment.

VI. RESULTS

Introduction

Several VLSI designs were used to determine the rate of success and performance for logic extraction and VHDL generation. Successive test cases involved VLSI design of increasing size. The reason for performing the test in this manner was to help limit the size of the problem space. This process localized possible errors and provided an examination of the efficiency of the logic extraction for increasingly larger VLSI designs.

At the time of publication of this thesis, VHDL 1076-1987, Intermetrics Version 1.0, did not fully support composite types and bus resolution functions reliably. Since most every signal type used the ten-level multiple-valued logic, every signal was a composite type. Furthermore, every signal had to be resolved through a bus resolution function. Due to the problems with the current VHDL 1076-1987 simulator, the VHDL V7.2 simulator from Intermetrics was used for validation.

Clock Generator

The first circuit tested was the clock generator. The clock generator consisted entirely of inverters, NOR gates, and a NAND gate. All components were implemented using static CMOS design. Therefore, no additional transistors remained after logic extraction. This example demonstrated that the transistor level simulation and gate level simulation of the clock generator in VHDL were similar in behavior. The inverters, NOR gates, and NAND gate were extracted correctly, and the behavioral descriptions of the inverter, NOR gate, and NAND gate were written correctly in VHDL. The clock generator simulated in VHDL but not in esim. This is an example of a VLSI design that will not simulate in a switch-level simulator due to the feedback loop in the ring oscillator.

Figure 30 is a circuit diagram of the clock generator represented at the gate level. The original transistor file for the clock generator contained 116 transistors. Figure 31 is the resulting component listing from the logic extraction process. The VHDL produced for the clock generator is shown in Appendix E. For the component listing of Figure 31, the inv(nA,nB,1) describes a static CMOS inverter with input nA and output nB. For the nor and nand gates, the input signals are listed within the brackets and the output appears by itself. Therefore, nor([nA,nB,nC],nD,1) would describe a static CMOS nor gate with nA, nB, and nC as inputs with nD as an output. Notice that the component listing corresponds with the circuit diagram of the clock generator. The resulting component listing of Figure 31 implies that only 42 transistors are required to fully describe the clock generator. Duplicate transistors were eliminated prior to generating the component output. Multiple transistors in parallel are commonly used to drive output circuitry. Logic extraction took 10 seconds of CPU time on a VAX 8800. Production of VHDL took 16 CPU seconds on the same machine.

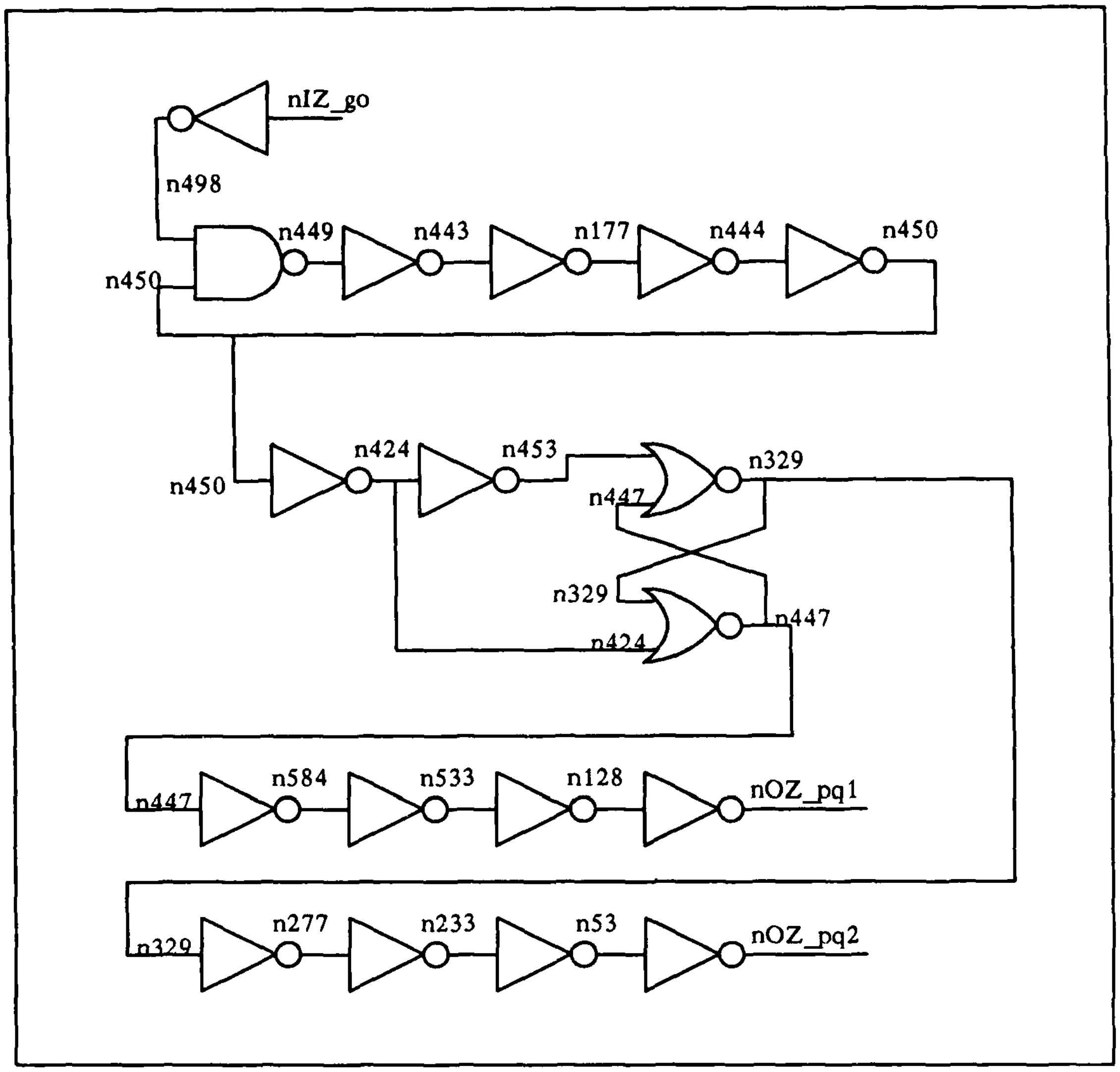


Figure 30. Circuit Diagram of the Clock Generator.

```
inv(n444,n450,1).
inv(n443,n177,1).
inv(n177,n444,1).
inv(nIZ_go,n498,1).
inv(n450,n424,1).
inv(n424,n453,1).
inv(n584,n533,1).
inv(n277,n233,1).
inv(n449,n443,1).
inv(n533,n128,1).
inv(n447,n584,1).
inv(n233,n53,1).
inv(n329,n277,1).
inv(n53,nOZ_pq2,1).
inv(n128,nOZ_pq1,1).
nor([n447,n453],n329,1).
nor([n329,n424],n447,1).
nand([n498,n450],n449,1).
```

Figure 31. Component Listing of the Clock Generator.

Register Macrocell

The register macrocell consists of D flip-flops, inverters, and multiplexers. These components were generated using static CMOS and transmission-gate logic. The D flip-flop is an example of a hierarchically generated component. The D flip-flop was generated from a transmission gate, high impedance inverter, and an inverter. As with the clock generator, no transistors remained after logic extraction. This example demonstrated that inverters, high impedance inverters, D flip-flops, and transmission gates were extracted correctly. The hierarchical extraction method worked correctly, and the behavioral descriptions of the inverter and D flip-flop were correctly written in VHDL. Like the clock generator, this design did not simulate in a switch-level simulator due to feedback in the D flip-flop.

Appendix B contains the resultant Prolog output for the 31-bit register macrocell that was extracted. A basic register cell for the 31-bit register is displayed in Figure 32. The labeling shown is also from the Prolog output in Appendix B. Figure 33 is a synopsis of the Prolog output that represents the register cell in Figure 32. Each cell has three inputs and one output. The gnd and GO signal are common to each register cell. The IZY30 signal is one bit of 31 bus lines that carry a 31-bit number. Depending on GO being high or low, either IZY30 or gnd will be loaded. There are also signals for phase 1 and phase 2 (PQ1 and PQ2, respectively) on the two D flip-flops (dff) for clocking the inputs and outputs. The two D flip-flops load on the inverse of the clock signal. The labels IN30, 1903 and 573 are internal labels that describe how the transmission gates, D flip-flops, and the buffer are interconnected.

There are 806 transistors in the original register macrocell. The final Prolog output listed in Appendix B contains 186 components. The extraction process took 54 seconds on a VAX 8800. VHDL generation took 22 seconds.

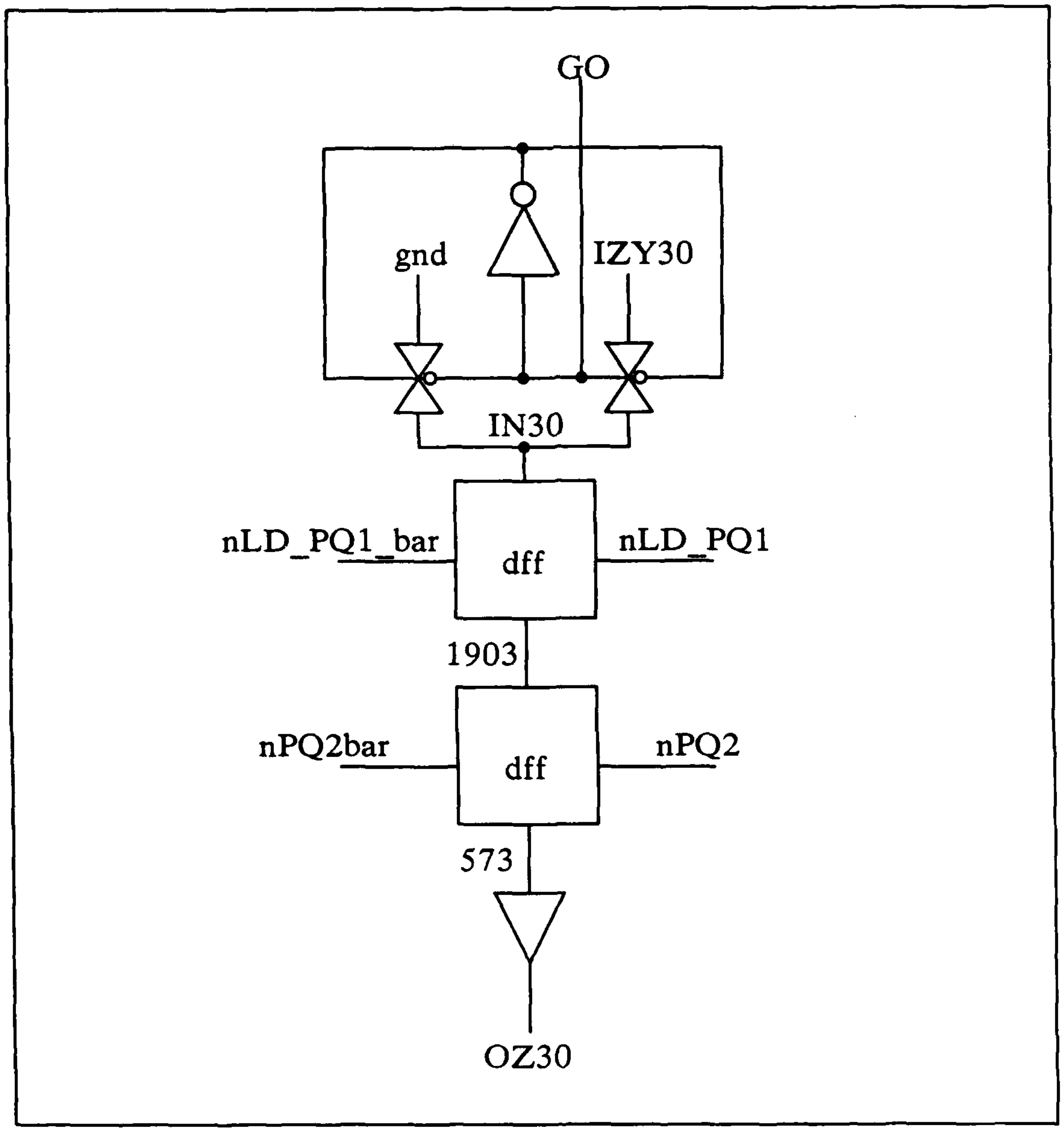


Figure 32. Basic Register Cell.

```
inv(nGO,nsbar30,1).
tgate(nsbar30,nGO,gnd,nIN30,1).
tgate(nGO,nsbar30,nIN30,nIZY30,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN30,n1903,1).
dff(nPQ2bar,nPQ2,n1903,n573,1).
buffer(n573,nOZ30,1).
```

Figure 33. Prolog Code for Register Cell.

31-Bit Carry Select Adder

The extraction of the 31-Bit Adder was accomplished in several passes. Figure 34 displays the work performed in each pass. In pass 1, the inverters, tgates, xor gates, and xnor gates were identified within the basic full adder cell. This pass exhausted the available knowledge within the extraction routine. A template of the full adder cell was generated from the Prolog information extracted. Figure 35 is the template of the full adder cell.

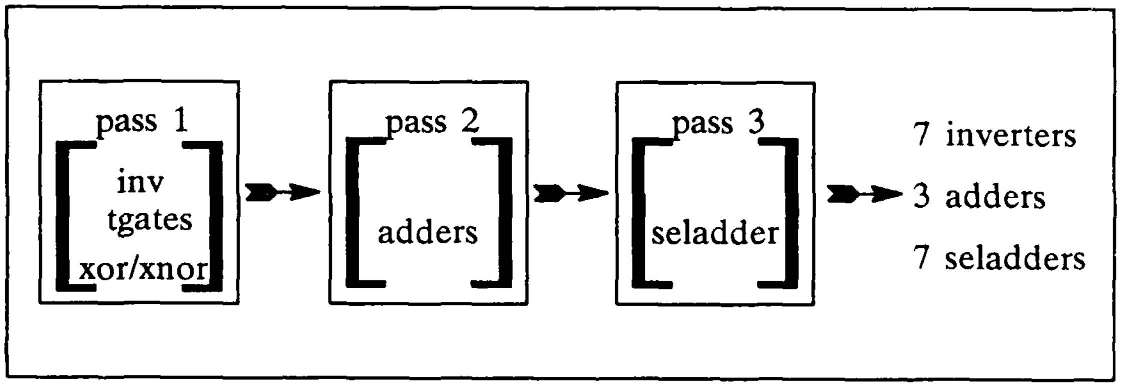


Figure 34. Extraction Procedure for 31-Bit Carry Select Adder.

```
adder :-
    xor(NA,NAbar,NB,NBbar0,Nxor,3),
    xnor(NA,NAbar,NBbar0,NB,Nxnor,3),
    tgate(Nxor,Nxnor,Nsumbar,Ncinbar),
    tgate(Nxnor,Nxor,Ncin,Nsumbar),
    tgate(Nxor,Nxnor,Ncybar,NBbar1),
    tgate(Nxnor,Nxor,Ncinbar,Ncybar),
    inv(Ncin,Ncinbar,1),
    inv(NB,NBbar0,1),
    inv(Nsumbar,Nsum,1),
    inv(Ncybar,Ncyout,1),
    inv(NA,NAbar,1),
    inv(NB,NBbar1,1),
    retract(xor(NA,NAbar,NB,NBbar0,Nxor,3)),
    retract(xnor(NA,NAbar,NBbar0,NB,Nxnor,3)),
    remove_tgate(Nxor,Nxnor,Nsumbar,Ncinbar),
    remove_tgate(Nxnor,Nxor,Ncin,Nsumbar),
    remove_tgate(Nxor,Nxnor,Ncybar,NBbar1),
    remove_tgate(Nxnor,Nxor,Ncinbar,Ncybar),
    retract(inv(Ncin,Ncinbar,1)),
    retract(inv(NB,NBbar0,1)),
    retract(inv(Nsumbar,Nsum,1)),
    retract(inv(Nsumbar,Nsum,1)),
    retract(inv(Ncybar,Ncyout,1)),
    retract(inv(NA,NAbar,1)),
    retract(inv(NB,NBbar1,1)),
    asserta(adder(NA,NB,Ncin,Nsum,Ncyouth,3)),
    (retract(inv(Nsumbar,Nsum,1))
    ;
    retract(inv(Ncybar,Ncyout,1))),
    fail.
adder.
```

Figure 35. Full Adder Template.

All of the labels were converted to variables by replacing the preceding "n" to "N". The template for the full adder cell was then appended onto the extraction procedure; and the next level cell, select adder, was extracted during pass 2. Figure 36 shows a block representation of how the select adder was constructed. The connectivity was verified for the select adder. The new template for the select adder was then appended onto the extraction routine.

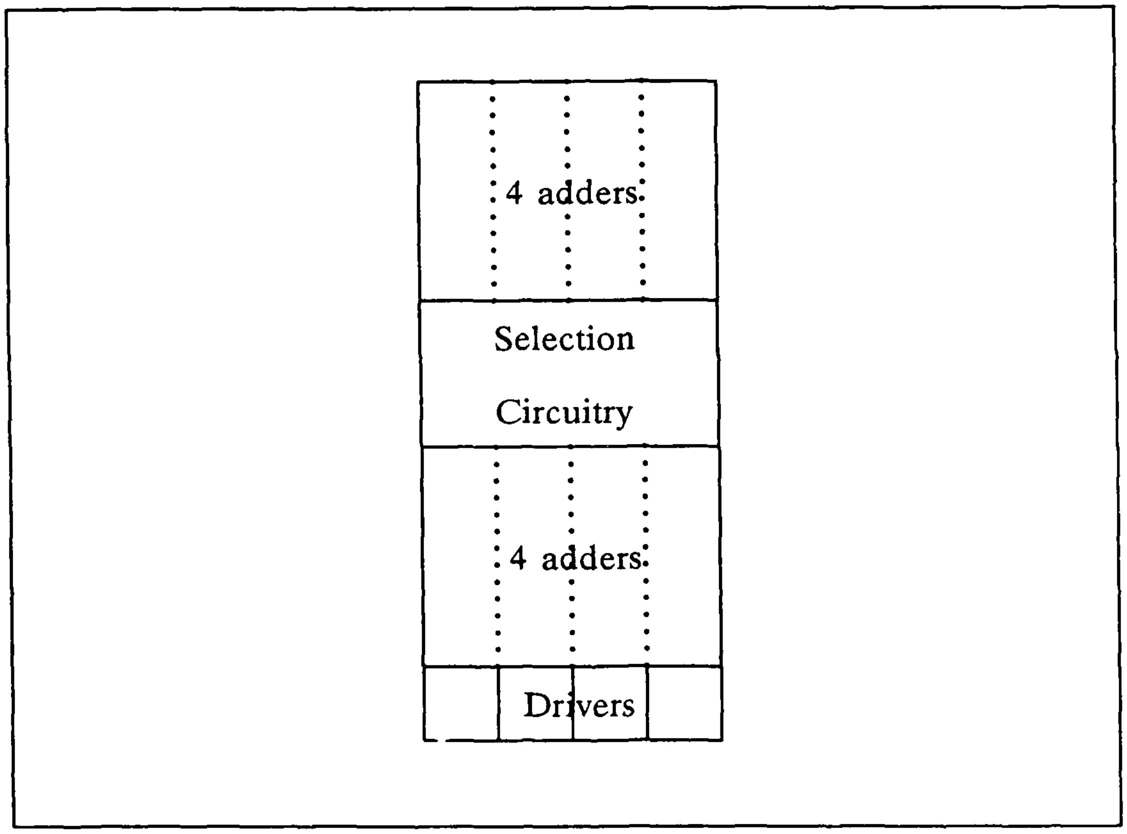


Figure 36. Select Adder Layout.

For pass 3, the entire 31-bit carry select adder cell was extracted using previously acquired knowledge extracted from lower level components. The resulting Prolog representation for the circuit is displayed in Figure 37. Figure 38 is a block diagram for the layout of the 31-bit carry select adder. The macrocell contains 2288 transistors. The total extraction process required 4 minutes, 33 seconds of CPU time on a VAX 8800. Producing VHDL from Prolog took 23 seconds.

```
inv(n2676,nIZ_cin_bar6,1).
inv(nOZ_sum1,n191,1).
inv(nOZ_sum0,n192,1).
inv(nOZ_sum2,n190,1).
inv(n190,nOZ_fsum2,1).
inv(n191,nOZ_fsum1,1).
inv(n192,nOZ_fsum0,1).
adder(nIZ_fa2,nIZ_fb2,n2967,nOZ_sum2,n2676,3).
adder(nIZ_fa1,nIZ_fb1,n2888,nOZ_sum1,n2967,3).
adder(nIZ_fa0,nIZ_fb0,nIZ_fcin,nOZ_sum0,n2888,3).
seladder(nIZ_fa6,nIZ_fa5,nIZ_fa4,nIZ_fa3,nIZ_fb6,nIZ_fb5,nIZ_fb4,
    nIZ_fb3,nIZ_cin_bar6,nOZ_fsum6,nOZ_fsum5,nOZ_fsum4,nOZ_
    fsum3,nIZ_cin_bar2,3).
seladder(nIZ_fa10,nIZ_fa9,nIZ_fa8,nIZ_fa7,nIZ_fb10,nIZ_fb9,nIZ_fb
    8,nIZ_fb7,nIZ_cin_bar2,nOZ_fsum10,nOZ_fsum9,nOZ_fsum8,n
    OZ_fsum7,nIZ_cin_bar5,3).
seladder(nIZ_fa14,nIZ_fa13,nIZ_fa12,nIZ_fa11,nIZ_fb14,nIZ_fb13,nI
    Z_fb12,nIZ_fb11,nIZ_cin_bar5,nOZ_fsum14,nOZ_fsum13,nOZ_
    fsum12,nOZ_fsum11,nIZ_cin_bar1,3).
seladder(nIZ_fa18,nIZ_fa17,nIZ_fa16,nIZ_fa15,nIZ_fb18,nIZ_fb17,nI
    Z_fb16,nIZ_fb15,nIZ_cin_bar1,nOZ_fsum18,nOZ_fsum17,nOZ_
    fsum16,nOZ_fsum15,nIZ_cin_bar4,3).
seladder(nIZ_fa22,nIZ_fa21,nIZ_fa20,nIZ_fa19,nIZ_fb22,nIZ_fb21,nI
    Z_fb20,nIZ_fb19,nIZ_cin_bar4,nOZ_fsum22,nOZ_fsum21,nOZ_
    fsum20,nOZ_fsum19,nIZ_cin_bar0,3).
seladder(nIZ_fa26,nIZ_fa25,nIZ_fa24,nIZ_fa23,nIZ_fb26,nIZ_fb25,nI
    Z_fb24,nIZ_fb23,nIZ_cin_bar0,nOZ_fsum26,nOZ_fsum25,nOZ_
    fsum24,nOZ_fsum23,nIZ_cin_bar3,3).
seladder(nIZ_fa30,nIZ_fa29,nIZ_fa28,nIZ_fa27,nIZ_fb30,nIZ_fb29,nI
    Z_fb28,nIZ_fb27,nIZ_cin_bar3,nOZ_fsum30,nOZ_fsum29,nOZ_
    fsum28,nOZ_fsum27,nOZ_cout_bar0,3).
```

Figure 37. Prolog Representation for 31-Bit Carry Select Adder.

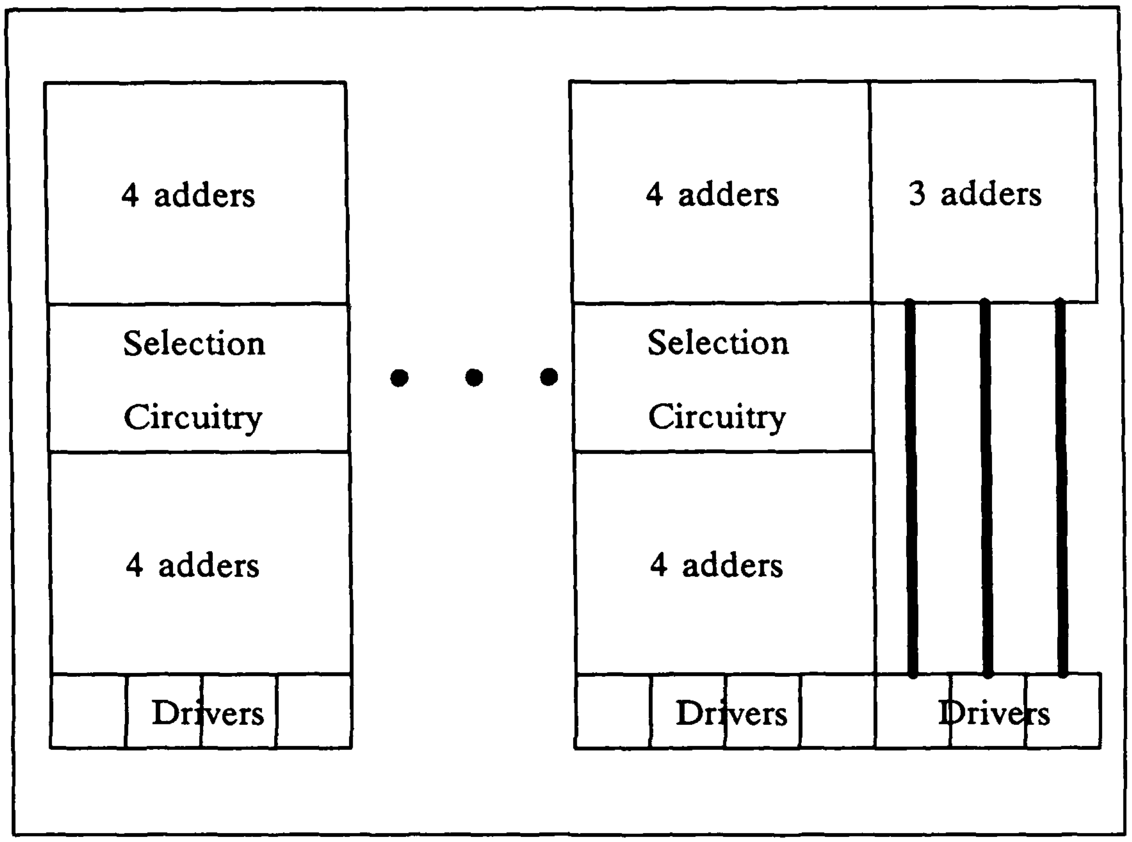


Figure 38. Block Diagram of the 31-Bit Carry Select Adder.

Multiplier Chip

The multiplier chip consists of a wide range of components, including many of those outlined in the previous designs. These components were largely generated using static CMOS, transistor, and transmission-gate logic. This example demonstrates the feasibility of the extraction process in a parallel environment and the functionality of the models for the p and n transistors with behavioral descriptions of extracted components. A thorough description of this chip was included in (Gallagher 1987).

The multiplier chip is a VLSI circuit consisting of 24,097 transistors. It was developed at AFIT in the AFIT CAD environment and was fabricated at MOSIS. The CIF file that was transmitted for fabrication was also used for the logic extraction portion of this project, to examine the utility of a Prolog extraction tool on VHSIC class components. Once the file was translated to sim format, the sim2pro routine was run on the multiplier chip sim file to generate the Prolog representation of the file. Upon first attempting to load the Prolog representation of the chip into Prolog on the SSC (A VAX system running Berkeley Unix at AFIT), an error of "Cannot recover from this error -- Bye!" was encountered.

The file was then decomposed into 5 files of 5,000 transistors each. Table 9 lists the CPU time required to process the files when 5,000 transistors were processed at a time and when the 5,000 transistor files were decomposed further into 1,000 transistor files and processed. The only components extracted on the first pass were inverters, transmission gates, clocked inverters, and D flip-flops.

Table 9. CPU Time to Process Multiplier Chip.

	Mult1	Mult2	Mult3	Mult4	Mult5
Transistors	5000	5000	5000	5000	4097
CPU time per 5000	177 min	97 min	107 min	101 min	92 min
CPU time per 1000	27 min	19 min	21 min	22 mon	24 min
Leftover Transistors	1880	295	699	483	638
Components Generated	1000	2298	2065	2181	1565

From Table 9, several points become apparent. The smaller the file operated upon, the less CPU time required for extraction. Furthermore, the process of extraction can be easily performed in a parallel environment with little or no interprocess communication. The only process intercommunication that may be performed involves merging left over transistor files for further extraction of components.

Transistors that constitute higher level components tend to be located within close proximity of each other. The tools for generating the switch-level representation of the layout design do so in a horizontal-first pattern. Within a given number of contiguous lines of a switch-level representation, there is a fairly high probability of interconnection between transistors. Therefore, dividing a large chip file into smaller transistor files will not severely affect the relative success of component extraction.

Only 83 percent of the multiplier chip was extracted on the first pass. This low success rate prompted a closer examination of the multiplier chip. Separate extractions were performed on the cell libraries that formed the multiplier chip. There were several nonstandard cell designs implemented on the multiplier chip. The nonstandard designs were a result of modifications to accommodate the esim simulator. Some of the cells were then extracted into Prolog templates. On a second pass of the multiplier chip, an extraction success rate of 94 percent resulted. The current STOVE_C program can only obtain an 84 percent success rate. No modification or addition of templates to STOVE_C can be accomplished without a considerable programming effort.

Table 10 contains the results from the logic extraction and VHDL generation for the multiplier chip on a VAX 8800. Figure 39 is a graph of the values within Table 10. All values were normalized relative to the total extraction time for the entire multiplier chip description in one file. The "Extraction of Leftovers" represents the relative time required to reextract all of the leftover transistors and components from the parallelized extraction process. The "Extraction per Processor" graph demonstrates the speedup due to breaking the multiplier chip description file into several subfiles. The domain of the graph may be considered as the number of virtual processors involved in the extraction process. The reason the "processors" label was chosen was that the separate files could have been processed on separate CPUs without any requirement for interprocess communication. The "Sum of Leftover and Average per Processor" represents the average time required to process the files plus the leftover time. This line may be considered as dividing the extraction process among n processors then collecting the residuals and extracting again. The final graph, "Linear

Speedup," is the relative speedup of O(n). The graph demonstrates the advantage of parallelizing the extraction problem.

Table 10. Extraction times for the Multiplier Chip.

Process Nodes	Average per Proc (Seconds)	Leftover Time (Seconds)	Average Normal	Leftover Normal	Total Normal
1	49331	0	1	0	1
2	7923	6073	0.160	0.123	0.444
4	2282	6167	0.046	0.125	0.310
8	582	6295	0.011	0.127	0.222
16	158	7714	0.0032	0.156	0.208
32	51.5	9669	0.00104	0.196	0.229
64	21	12289	0.000425	0.249	0.276
128	12.5	15894	0.000252	0.322	0.355

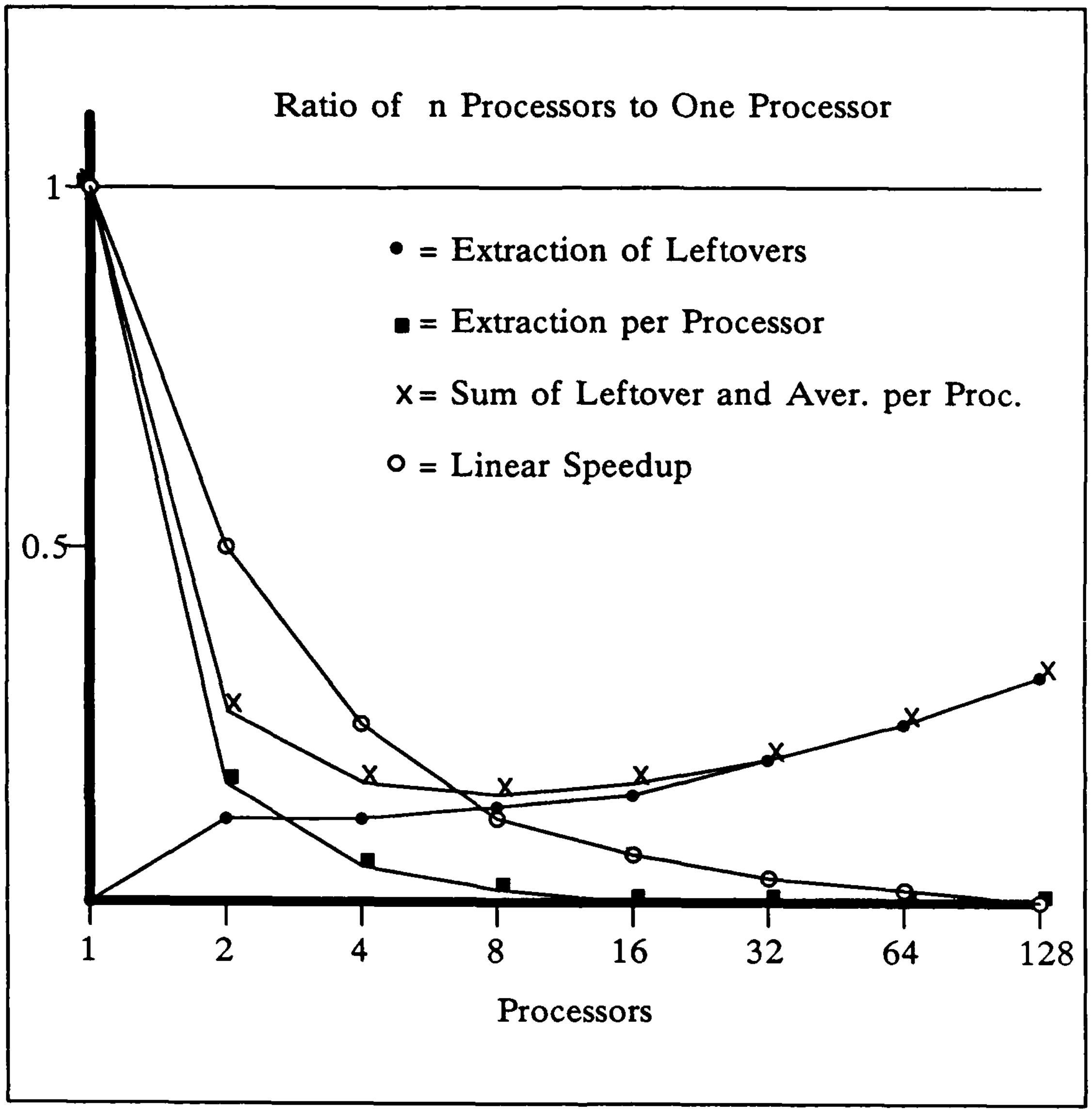


Figure 39. Graph of Relative Speedup for Increased Processors.

VII. CONCLUSIONS AND RECOMMENDATIONS

Conclusions

Prolog is considered to be a less efficient language than Lisp. Functionally, anything performed in Prolog may also be accomplished in Lisp. In order to increase the efficiency of the extraction, a Lisp form of the tool should be developed using the proven concepts from the Prolog code. As a rapid prototyping tool, the use of Prolog permitted production of a working extraction system within less than two weeks. For this and similar problems, Prolog provides an excellent software tool for quickly identifying requirements and generating a proof of concept.

The nature of the pattern matching problem adapted quickly to a parallel solution. No interprocess communication was required, providing for a completely decoupled processing environment. The original extraction problem in Prolog was close to O(n!). Parallelization of the solution then allowed for greater than O(n) speedup. This speedup result was only realizable from the high rate of pattern matching success within each divided portion of the chip design. However, the rate of speedup success did not continue to increase as the number of processors increased. This was attributable to a decrease in the rate of matching success as each portion of the chip design became smaller. At this point no conclusions can be made as to the relation between the number of transistors per processor versus speedup and the number of processors versus speedup.

Work with the multiplier chip shows that a "blind" extraction of an unknown chip is an unreasonable task without intervention from a VLSI design engineer; however, if the extraction tool is used during the layout process of a chip, the layout of the design can be verified directly against the VHDL structural specification for the chip. Figure 40 portrays how the extraction tool would be used during the layout process.

Figure 40 shows that the lowest level of layout on a custom VLSI cell would be used by the extraction tool to identify the low-level components. The design engineer would verify the components and their interconnectivity based upon graphical or textual feedback from the extraction tool. The template generated from the cell would then be used to extract and verify the higher level macrocell generated. This process would then be performed iteratively until the hardware description is completed. The result of the extraction process would generate a VHDL structural description of the layout. The derived VHDL structural description would be used for testing, verification against the original VHDL structural specification, and as the actual VHDL structural description of the hardware. A recommended thesis including this concept will be discussed in the recommendations section of this chapter.

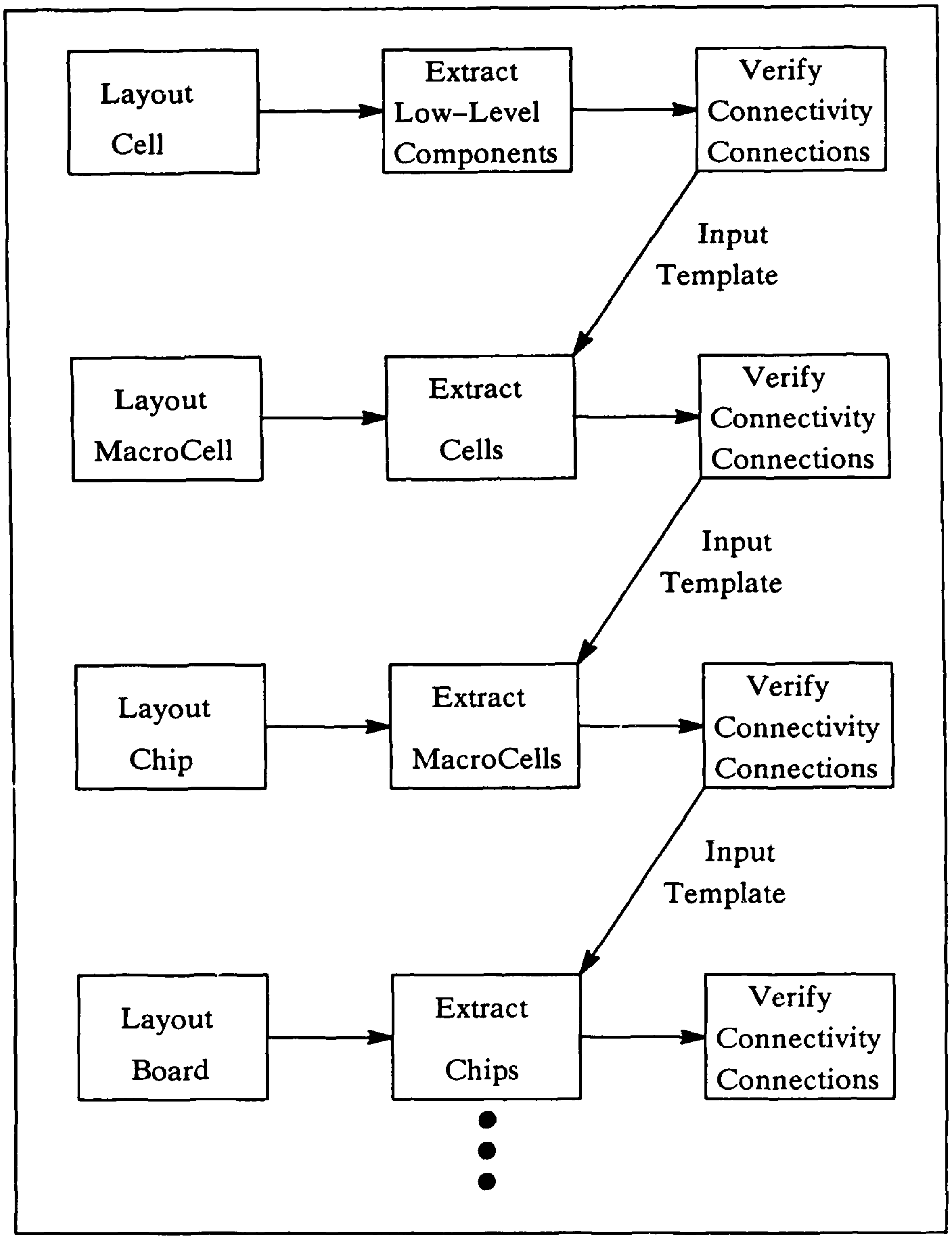

Layout Cell
Extract Low-Level Components
Verify Connectivity Connections
Input Template
Layout MacroCell
Extract Cells
Verify Connectivity Connections
Input Template
Layout Chip
Extract MacroCells
Verify Connectivity Connections
Input Template
Layout Board
Extract Chips
Verify Connectivity Connections

Figure 40. Layout Verification Process.

For the problem of providing a VHDL structural specification for a chip without an original VHDL description, this tool would be used in an identification and extraction mode by a design engineer. For those designs with nonstandard transistor configurations, the ten-level multiple-valued logic system in VHDL would simulate the leftover transistors. The result would be a VHDL specification generated for any CIF formatted chip representation.

For multiple-valued logic systems, specification of the values is insufficient for proper modeling. The algebra for manipulating the logic values is crucial to the correct execution of the simulation model. Without a definition for a specific algebra, the same multiple-valued logic system could be used for the same hardware specification but produce vastly different simulation results.

The designs used in this thesis, except for the multiplier chip, were extracted and simulated in VHDL V7.2. The designs simulated correctly. The clock generator was simulated at the component and transistor level with similar simulation results. Due to numerous bugs that existed in the recently released Intermetrics VHDL 1076-1987 V1.0, these models would neither model generate completely nor would they initialize during simulation. These problems were documented and submitted to Intermetrics as legitimate software problems with the analyzer, model generator, and simulator. Considering the success in extraction and simulation in VHDL V7.2, the ten-level multiple-valued logic system and Prolog-based logic extraction tool have made a significant contribution to the VLSI design process.

Recommendations

From the work performed in this thesis, the benefits of applying knowledge-based systems to the generation of VHDL for VLSI designs have become apparent. One portion of a formal verification environment has been identified within the AFIT CAD cycle. Figure 41 demonstrates a hypothetical view of a directly related thesis effort that should be pursued from this work. Additional work on an intermediate representation called Symbolic Intermediate Form would help to identify requirements to incorporate the use of logic extraction, formal verification, schematic output, analysis of VHDL, testing, and production of VHDL. The SIF would also be simulatable, thereby allowing for the automatic generation of test vectors (Clocksin 1987). Additional information within sim may be used to help establish positional data, timing information, and capacitive loading. The positional data would be kept throughout an extraction process so that a graphical interface would position components relative to their position in the VLSI layout design. Timing and capacitive information could be extracted from the transistor sizing data and capacitive data in sim in order to produce more accurate VHDL timing models. Finally, a future implementation of this system would provide a generic CAD tool that would allow for conversion to VHDL for any VLSI CAD environment in industry or university.

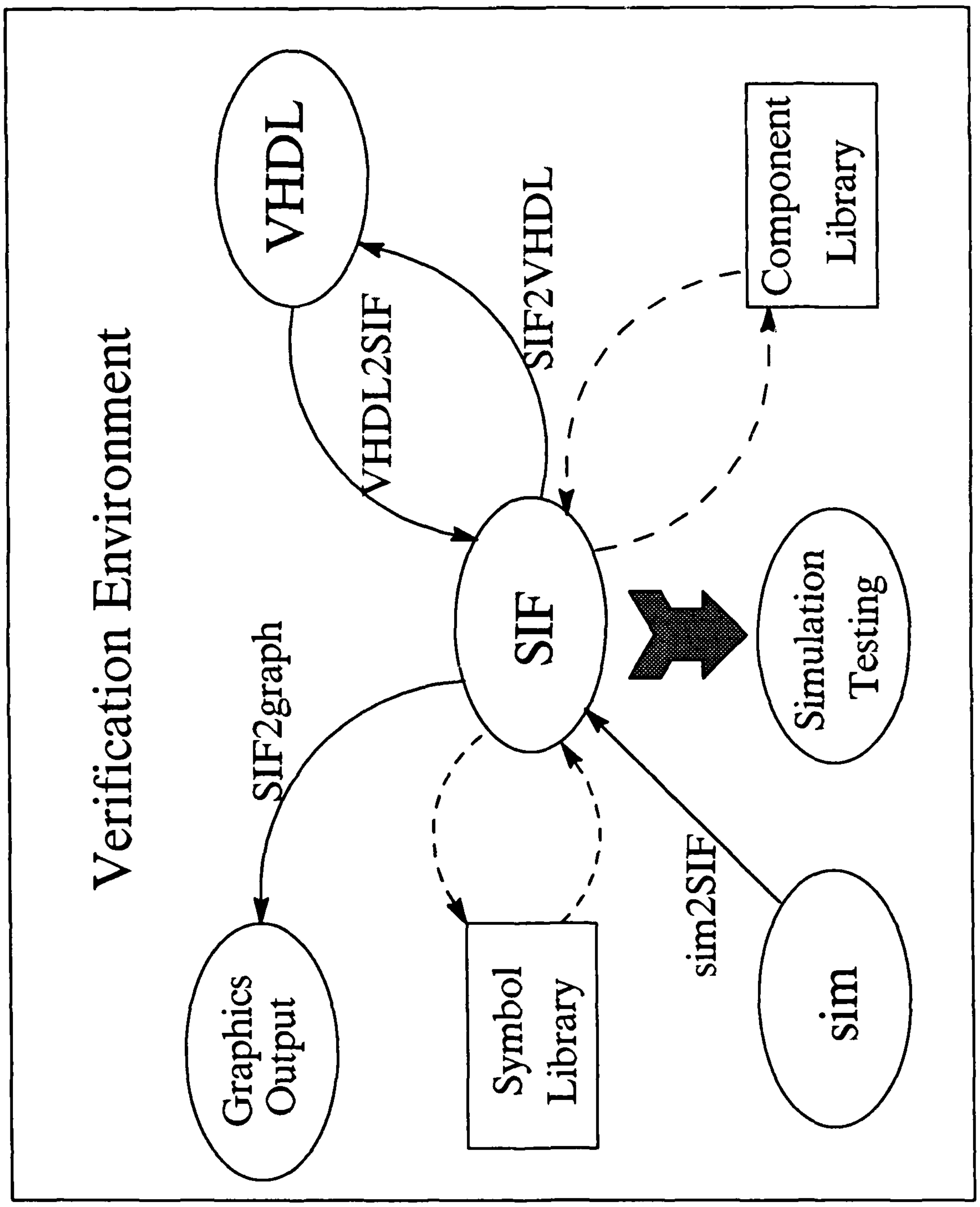


Figure 41. Formal Verification Environment.

A follow-on thesis to develop a generic VHDL extraction tool should be based, abstractly, on the system shown in Figure 41. Instead of a specific sim2SIF routine to convert the sim transistor netlist format to Prolog clause form, a generic tool should be written that would query the design engineer for a field definition of the transistor netlist format to be used. Through the extraction process shown in Figure 40, the design engineer would begin to build a library of templates to describe the basic cells in the standard cell library. Before a template would be built, SIF2graph would display the component configuration using as many of the "known" components to exhibit the composition and connectivity of the selected VLSI cell. Once the design engineer verifies the composition and connectivity, a new template would be generated and a graphical symbol selected for storage in the Symbol Library. The design engineer would be provided the option to have a VHDL description generated for the cell through SIF2VHDL or to enter a VHDL behavioral description manually. The VHDL for the new component would then be entered into the Component Library. The design engineer would also have the option to request that a set of test vectors be generated from the template.

The above description of a follow-on thesis would provide a means for translating an existing CAD library to VHDL. The tool would be useful to both industry and universities. Through the use of Prolog, the tool would be readable, simple, easy to modify, and easy to maintain. Furthermore, the system shown in Figure 41 would remove many impediments to the acceptance of VHDL 1076-1987 as a standard hardware description language.

APPENDIX A: Annotated Bibliography for VHDL

The purpose of this appendix is to provide a brief list of some background material for this thesis and to provide a summary of some VHDL informational sources. Therefore, this appendix is divided into two parts. The first part furnishes a list of several helpful sources the reader may wish to review prior to reading various portions of this thesis. The second part lists summaries of several important references regarding VHDL.

Background Sources

Listed below are the subject areas encompassed within this thesis. Included with the subject areas are sources that provide worthwhile information.

VHDL 1076-1987:

IEEE Standard VHDL Language Reference Manual, IEEE STD 1076-1987.

Hardware Simulators:

Rubin, Steven M. *Computer Aids for VLSI Design*. Reading: Addison-Wesley Publishing Company, 1987.

VLSI Layout:

Weste, Neil and Kamran Eshraghian. *Principles of CMOS VLSI Design*. Reading: Addison-Wesley Publishing Company, 1985.

Prolog:

Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. Wokingham: Addison-Wesley Publishing Company, 1987.

Clocksin, W.F. and C.S. Mellish. *Programming in Prolog*. New York: Springer-Verlag, 1987.

C Programming Language:

Waite, Mitchell, Stephen Prata, and Donald Martin. *C Primer Plus.* Indianapolis: Howard W. Sams and Company, 1987.

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* New Jersey:Prentice-Hall, Inc., 1978.

Predicate Calculus and Abstract Algebra:

Chang, Chin-Liang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving.* Boston: Academic Press, 1973.

Stanat, Donald F. and David F. McAllister. *Discrete Mathematics in Computer Science.* Englewood Cliffs: Prentice-Hall, Inc., 1977.

Annotated Bibliography

The format of this annotated bibliography lists the bibliographic citation followed by a summary of the reference. The annotated bibliography is listed in alphabetical order by author.

Aylor, J.H. "VHDL - Feature Description and Analysis," *IEEE Design and Test,* 3: 17-27 (April 1986).

Eight hardware description languages were examined based on specified criteria. Some worthwhile terms were extracted and noted here. Behavioral abstraction: definitions of algorithms at a high-level with respect to lower-level constructions of primitives. This allows for adders, multipliers, etc., to be described in terms of algorithms instead of low-level logical functions of AND, OR, NAND, and NOR. Structural hierarchy: allows for construction of high-level primitives at any level in the design.

The following figure with definitions was worth reconstructing from the article.

```
                    Design Description
                            |
              --------------------------
              |                        |
  Combinational Function     Sequential Function
                                       |
                          --------------------
                          |                  |
                     Synchronous       Asynchronous
                        Logic              Logic
```

The following terms were defined in the article:

(1) Combinational Function: dependent solely on the present input values for its output values (No memory of prior events).

(2) Sequential Function: output dependent upon current input values and prior events (Memory of prior events).

(3) Synchronous Logic: Clocked.

(4) Asynchronous Logic: changes at one stage affect the next stage without any external synchronization.

(5) Generic Structures: components that can be defined by user-set parameters (M N-bit adders may become four 8-bit adders.

The language should supply documentation for all levels and simulation for the appropriate level. The language should also allow for specification of synchronous (clock driven or cycle driven) or asynchronous (event driven) logic. Thus, simulations can become mixed-level and mixed-mode. It appears that the hardware description language must possess some of the same salient features as Ada.

Additional terms that were described in this article are listed:

(1) Scope-range of hardware design: Digital-systems, Gate-level, Digital-circuit, Combinational, Synchronous, Asynchronous, and Mixed-mode.

(2) Management of design: Hierarchy, Modularity, Incremental compile, Arbitrary decompose, Libraries, Data abstraction, User type, Alternate description, and Reusable designs.

(3) Timing description: Timing at all levels, Specify timing data, User-defined data, Timing constraints, Propagation delay, Inertial and transport.

(4) Architectural description: Algorithmic, Architectural, Parallelism, Control and data separate, Descriptive continuum, User assertions, Explicit instructions, Implicit structure, Generic components, Regular structures, Recursive structures.

(5) Interface description: Explicit interface, defined at all levels, functional equivalence, strongly typed interface.

(6) Design environment: environment info.

(7) Language extensibility: User-defined data types, Design tool support, Multiple technologies, multiple methodologies.

d'Abreu, Michael. "Gate-Level Simulation," *IEEE Design and Test*, 2: 63-71 (December 1985).

There are four methods for defining delay models, transport, rise/fall, ambiguity, and inertial. The transport delay does not show an accurate portrayal of the circuit's timing performance. Rise/fall allows for specification of the rise and fall times of the output signal. Ambiguity delay model deals with a range of values for a signal change. Inertial delay examines the minimum pulse width required to propagate a change in a succeeding gate. Multiple values can be used to express a signal in reference to several states, 0, 1, or high impedance.

There are two basic types of simulators, compiler-driven and table-driven, event-directed. The compiler type is among the earlier simulators. The simulation is compiled into directly executable machine language. Compiler-driven simulators have a disadvantage in that they do not test for race or hazard conditions. Therefore, these simulators are suited to simulating synchronous logic.

Since most digital circuits are usually 10-15 percent active, there is no real need to simulate the whole device. Table-driven, event-directed simulators only are concerned with those circuits that are active.

DeGroat, Major Joseph W., Instructor of Electrical and Computer Engineering. *Video Tape Lecture.* "*VHDL Design Entities.*" *School of Engineering*, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987.

VHDL Design entities consist of an interface and design entities. The interface declaration is made by the reserved word "entity," followed by a design entity name, port declarations, and an end reserve word. The port declaration consists of the reserved word "type" and a mode. Modes of type "in," "out," and "inout" may be described as:

in - driven outside and read in the design entity.

out - driven by the entity and read out of the design entity.

inout - bidirectional.

buffer - driven inside the entity and may be read inside or outside the design entity.

linkage - may not be driven or read. This is used for cases where signal flow is not known.

Generic declarations define constants where values are determined by the environment when a design entity is used. Generic declaration may include a default expression.

Alias - gives another name for an object.

Constant - defines a constant value.

Type - defines the type of an object.

Assert directive - specifying required operating conditions and characteristics of the entity.

Initialize directive - specifies initial values of objects.

Body descriptors specify what goes on inside the design entity. The body may be behavioral or structural. The body is specified by the reserved word "architecture" followed by the name, "of," and the linkage name to the interface described by the body. Structural specifications describe the entity in terms of gate-level specifications. Behavioral specifications describe the entity in terms of algorithms and assignment statements. No components are instantiated within a behavioral specification. Use of the 'CHILDLESS attribute will allow checking for component instantiations within an architectural specification ('CHILDLESS = TRUE indicates a purely behavioral specification). Mixed-mode is a combination of both. More than one body may be used to describe an entity. The configuration body is at the root of the tree in a hierarchical structure of design entities. Within the configuration body, a block statement followed by component instantiations of lower level design entities describe the root design entity. Hierarchical structures are realized through component instantiations at the root.

Binding indication binds one port to another through instantiations. Generics may also be used to temporarily delay specification of where an entity may reside on a board. Behavioral description contains no components and is a self-contained description. It specifies data flow and timing delays through assignment statements. Mixed bodies contain component instantiations and assignment statements.

Dewey, Al and Anthony Gadient. "VHDL Motivation," *IEEE Design and Test*, 3: 12-16 (April 1986).

As the title suggests, the article provides the reason for the creation of VHDL. The need for documentation that is detailed, up-to-date, and accurate is addressed. VHDL should also decrease design time and cost in an area where off-the-shelf ICs cannot meet the demands of today's systems. Generic design components within VHDL allow for repeated use of previous design investments, providing for more efficient management of the design process. VHDL uses the specification-and-body concept of Ada, thereby providing for the interface specifications to be represented separately from the associated bodies. This allows for top-down, as well as bottom-up, management. Industry currently provides excellent tools within specific areas. Very little interfacing between these tools is possible without the use of special data conversion software. Thus, the need for an open-system design architecture. Semicustom and application-specific ICs are going to become increasingly popular by 1990. VHDL must provide clearly defined interfaces, provide user documentation, facilitate design second-sourcing, and protect proprietary design system elements. Design is falling short of the fabrication technology. A hardware description language is necessary to shorten this gap. VHDL allows for examination of the design process from a structural and behavioral point of view. The structural design is concerned with the register-transfer level up to microprocessors, memories, etc. Behavior-oriented design encompasses the boolean expressions, the algorithms, followed by the system input/output specification. This enables consideration of software and hardware within the design process.

Gilman, Alfred S. "VHDL - The Designer Environment," *IEEE Design and Test*, 3: 42-47 (April 1986).

This article states that TI designed a VHDL simulator. There are no references to any documentation. VHDL support environment consists of six components. There are five tools, the analyzer, the reverse analyzer, the design library manager, the simplifier, and the simulator. The final component is the design library. The Design Library contains the intermediate VHDL attributed notation (IVAN) representation for design entities. Also contained within the Design Library are the relationships between the Package Procedure or Function, Interface, Architectural Body, and Configuration Body. Other information pertaining to "contexts" is maintained. The Design Library Manager is the software that supports accessing the design library. The VHDL simulator may be thought of as a collection of test bench equipment, data stream playback, signal generator, test bench control data stream playback, clock, bed of nails, logic analyzer, and data recorder. The simulator consists of simulation model generator, the kernel, and report generator. The model generator takes design library information and converts it into an Ada program. The kernel incorporates the Ada program, workbench tools, and necessary simulation required constructs.

The analyzer checks for static design errors and translates VHDL text to IVAN. The reverse analyzer takes IVAN data in the design library and converts it to VHDL form for any design entity.

Intermetrics, Inc. *VHDL User's Manual: Volume 1 Tutorial.* U.S. Air Force Contract F33615-83-C-1003. Bethesda, Md., 1 August 1985.

Chapters 2, 3, 5, 6, 7 of this manual explain how the constructs of VHDL are used in simulations. Volume I is divided into Introduction, followed by modeling of hardware structures, asynchronous & synchronous circuits, and complex behaviors. The introduction identifies the design entity as a collection of information specifying a hardware component. The design entity is made up of an interface description and a collection of body descriptions. Interface descriptions are described in terms of input and output ports reflecting their logical function to the outside world. A port declaration has an associated mode and type. Mode indicates direction of flow. Types include user-defined and predefined. The keyword, entity, begins an interface descriptor, followed by end.

Interface descriptions can be looked at as "black-box," whereas body descriptions can be looked at as a "glass-box" or "clear-box" view. The header, architecture, followed by block specifies a body description. Three figures are offered examining body descriptions at a gate-level, RTL-level, and procedural-level. These figures may also be viewed as gate-level, data-flow, and algorithmic descriptions.

In chapter 2, Modeling Hardware Structures, defining components, defining unidirectional data paths, defining bidirectional data paths, and interconnecting components is discussed. The objective is to be able to interconnect components in modeling hardware structures. The component declaration has a lot in common with the interface descriptor, entity. It is used to define ports for subcomponent interconnections. When a component is declared, the design entity for that component need not exist within the library until simulation, since binding does not occur until that time.

Unidirectional signal paths may be thought of as single-source. Bidirectional signal paths may be thought of as multi-source. Signal paths are characterized of the same types as ports, but unlike ports, they are not labeled with a mode to show direction. The means by which single-source and multi-source are differentiated is by using tristate logic for multi-source. Furthermore, the signal is labeled as a bus and a bus resolution function is used to determine which value is output given several different source's input. The component statement provides generic descriptions of the design entity. Component instantiation occurs using the component name with the port reserved word.

Chapter 3 on Modeling Asynchronous Circuits discusses several topics, describing data transforms, simple signal assignment statements, conditional signal assignment statements, and selected signal assignment statements. This function becomes necessary when a hardware device that is needed from the library is undefined and the internal behavior of the device is known. One or more signal assignment statements make up a data transform. Signal assignment statements describe a function to be performed over time. Simple signal assignment statements consist of a target, assignment operator, operands, function to be performed, and the time it will take to perform the function (S<= A and B after 10ns;). This is worth considering in scheduling events, possibly, for a simulator. Conditional signal assignment statements allow for a choice of one of several functions that may be evaluated to produce a result. The simplest conditional signal assignment statement with one function is a signal assignment statement. The selected signal assignment statement is similar to the Ada case statement in that a control expression is used to select a function to perform a signal assignment (with exp select... on p. 3-8 of tutorial).

In chapter 4, modeling synchronous circuits is discussed in three sections, block statements and guards, memoried signal assignment statements, memoried-conditional signal assignments, and memoried-selected signal assignments. The goal of this chapter is to allow for specification of a control signal that will determine when a selected set of inputs can effect a selected set of outputs.

The block consists of a declarative part and a statement part. A guard expression is inserted after the block statement. The guard expression only affects those signal assignment statements within the statement portion that are designated with the "memoried" reserved word. Only when the guard condition is TRUE will the memoried assignment statements be executed. Other assignment statements within the guarded block that do not contain a "memoried" reserved word will be evaluated when their respective inputs are changed. Attributes may also be used to further describe guards. The use of an attribute may allow for synchronization to occur on the edge of a signal (i.e., when clock'stable = FALSE). Conditional and memoried statements may be used together.

Chapter 5, Modeling Complex Behaviors, sequential statements and the process statement, enabling and disabling inputs, and using process statements. Process statements respond to events. Execution of a process statement may schedule a new event. An event occurs when a signal changes. The process statement description consists of the reserved word "process," followed by a sensitivity list (events that invoke the process), a declarative portion for local variables, assignment statements with control-flow operations, and the reserved words "end process." Process statements may also be used to enable and disable inputs. Use of process statements helps speed up simulations by minimizing the scheduling of events.

Chapter 6 covers the use of types and objects. The sections covered are types and subtypes, classes of objects, buses, and atomic vs granular signals. The type definition is similar to the Ada construct with the Ada strong typing concept. There are three type of objects: constants, variables, and signals. Signals are different from variables in that signals contain a collection of "containers" that may contain values. Signals are only modified by signal assignment statements and their value-assignments are scheduled to occur at some event. Signals may also have multiple drivers. The use of the reserved word "atomic" when declaring a signal type specifies that all subelements must be treated as a whole. For simulations, it is not necessary to monitor all of the subelements for change since they can only be changed as a whole. Granular signals can have their subelements modified independently. Signals that are buses are required to be atomic. These signals have multiple drivers and use a bus resolution function to convert an unconstrained array of input values into a single output value.

Chapter 7, Manipulating Objects, explains some of the aspects of how a simulation proceeds. Sequential and concurrent statements are the two basic execution sequences of statements. Sequential assignment statements may perform variable (single target ":=") or signal (multiple target "<=") assignments. Variable assignments may have only a single evaluation expression. Signal assignments may have several evaluation expressions. Delta delay is defined as an infinitesimally small delay (after 0ns). A process statement consists of a collection of assignment statements that execute in response to an event. An event is the change in value of a signal. ''rocess statements also define drivers for signals. It is best to limit the scope of a process to driving only one signal. Conditional assignment statements can be used to express the same information as process statements when it is necessary to emphasize the bare essentials of the data transform. Execution cycles are used to repetitively compute the new signal values. Since simulations are considered to begin at some "0" time, initialize directives provide a starting point for all signals.

Each simulation cycle consists of the following three steps.

1. A signal is reevaluated if it is driven by a driver whose value has just changed.

2. A process statement is executed if a signal to which it is sensitive received a new value in step 1. Such process statements may execute signal assignment statements in order to schedule new future values for drivers.

3. The global simulation time advances until the values of one or more drivers change.

Step 1 of the first simulation cycle is never executed. Thus, step 2 of the first simulation cycle is never executed. (Further exception) If there exist process statements not sensitive to anything, they will be executed during step of the first simulation cycle. Simulation cycle will repeat until there are no longer any future events scheduled or a control stops execution.

Chapter 8 on Design Entities covers much of the information presented in the video tape lecture entitled, "VHDL Design Entities," presented by Major DeGroat. The video tape lecture is listed toward the end of the bibliography.

Chapter 9 discusses the networks and their evaluations. For a given signal there exists a root node and possibly several drivers and or sinks. A bus resolution function is necessary when there are several bus drivers. The values of the drivers are propagated up to the root. Afterwards, the value in the root is propagated down to the sinks.

Chapter 10, abstraction capabilities, includes information on user-defined types, packages, and subprograms. The user-defined type may be of the scalar or composite type. Composite types may be made of records or arrays. Strong typing is performed much like Ada. Subtypes may be used to limit the range of type. Subprograms execute a series of instructions in zero simulation-time. A subprogram may be a function or a procedure. Packages are used for grouping and storing declarations. This allows for sharing between designers. All of the items discussed above in chapter 10 may be included within a package. Packages are made visible much like Ada.

Advanced descriptive capabilities are covered in chapter 11. Areas covered are attributes, assertions, and generation statements. Attributes are similar to attributes in Ada except that they are associated with an entity and may be predefined or user-defined. Assertions allow for operating conditions and characteristics to be checked. Different levels of severity maybe associated with the assertion for reporting violations during the course of the simulation. The generate statement allows for the generation of multiple instantiations of a component much like a macro in assembler.

Appendix A covers the VHDL Support Environment and a discussion of its associated parts. A.4 gives a discussion of the simulator and its similarity to the test bench concept. Appendix C contains a glossary.

Linderman, Captain Richard, Assistant Professor of Electrical and Computer Engineering. Video Tape Lecture. "Types and Objects in VHDL." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987a.

Use of types and objects in VHDL allows for error detection to prevent mismatches between different interface types. It forces implementation of special functions to convert different types. Types are composed of scalar and composite. The scalar types are enumerated and numeric. Numeric types may be further divided into integer, floating, and physical. Physical types maybe time, power, or any other measurable physical types. The composite types are composed of arrays and records.

Objects are either signal (associated with time), variables (only current time), and constants. Some examples of real and integer subtype declarations were given. Some predefined types are bit, bit_vector, integer, real, boolean, character, and string.

The classes of signal are Atomic and Granular. Atomic signals are assigned as a whole and may be a bus (signal with multiple drivers). Granular signals are partially updated (separately addressed) and are not a bus. Examples of granular signals were presented. The keyword "Atomic" denoted an atomic signal. If the keyword is not there then it is granular.

A bus resolution function must be used when a signal has multiple drivers. Declarations may be explicit or implicit. When explicit, they are declared in the declarative part. When implicit, they are declared in the block guard, loop, or generate statements. Using static in a variable declaration allows for remembering it between process invocations. Processes, functions, and procedures contain only sequential statements. Concurrent statements may be block statement, process statement, etc.

Linderman, Captain Richard, Assistant Professor of Electrical and Computer Engineering. Video Tape Lecture. "Modeling Circuits in VHDL." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987b.

An expression may be used for logical, relational, or arithmetic operations. Logical expressions are more bit level (XOR, NOR, AND, etc.). Relational expressions are used to describe control flow. Arithmetic expressions are used for integers and floating point operations. An array expression is concatenation, indexing, or slicing. Static expressions are evaluated at analysis time. Generic expressions are evaluated at the beginning of the simulation. Dynamic expressions are evaluated during the simulation.

Signal assignment statements schedule a transaction to occur. A possible signal assignment statement: "Output <= IN1 and IN2 after 5ns;" The delta delay is used when 0ns is used or no after clause is specified. Block statement constructs were also covered.

Linderman, Captain Richard, Assistant Professor of Electrical and Computer Engineering. Video Tape Lecture. "VHDL Process Statements." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987c.

A good summary of the simulator execution cycle may be found within the course of this tape. The topics covered in this lecture are signal assignments, simulator execution cycle, and process statements. Process statements are used to optimize simulator performance. Signals have a time dimension (remember past values). The value of a signal is determined by drivers. A transaction consists of two dimensions, time and value. Transactions are used for updating the appropriate driver queue. A signal may be specified by either a transport clause or as an inertial signal assignment. Only when the actual signal has changed does an event occur.

Evaluation cycle for a non-bus signal:

1. A non-bus signal has only one driver.

2. As time advances, a transaction may occur on the driver.

3. The value of the driver is assigned to the signal.

4. A change in the signal value causes an event on that signal.

5. Process statements respond to events.

Process statements have sensitivity parameters specified through a list of signals. Evaluation cycle of a bus signal:

1. A bus signal has multiple drivers.

2. As time advances, a transaction may occur on a driver of the signal.

3. The values of all of the drivers of the signal are resolved together.

4. The resolved value is assigned to the signal.

5. Events are propagated as before.

Execution cycle:

1. Time advances.

2. Signal values are evaluated.

3. Values are assigned to signals.

4. Process statements are executed.

5. New signal values are projected.

Process statements are "black-box." Simulator optimization can be accomplished through efficient use of guards and process statements for enabling and disabling process execution.

Linderman, Captain Richard, Assistant Professor of Electrical and Computer Engineering. Video Tape Lecture. "VHDL Examples." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987d.

This tape covered a top-down description of an ALU, SIPO, and a CMOS static latch. The use of gate, logic, board, and system to describe the level of abstraction were used as attributes for entities. When combining different types in expression evaluations, parentheses may be used for matching. If two variables of type power were divided by each other the result would be no unit of measure. It could then be used to be modified a delay factor in units of time. Failure to use the parentheses would cause an error. A binary attribute (B'0000) was used as identify and modify a vector. The language objectives are as follows:

1. Support for hardware techniques.

2. Support for design styles.

3. Support for hardware design methodologies.

4. Support for design automation tools.

5. Support for management of design data.

Lipsett, L., E. Marschner, and M. Shahdad. "VHDL - The Language," *IEEE Design and Test*, 3: 29-41 (April 1986).

A design entity is the principle hardware abstraction. It provides for separation of interface and function. Design Entity => Interface + Body(s). The interface is the pinouts, timing assertions, and other factors needed to be known by component users. The body is the organization and/or operation of a component (behavior, structure, register-transfer ops, etc.). Interface => Ports + Generics. Body => Architectural Bodies + Configuration Bodies. Architectural bodies describe how the input and output ports of a design entity relate. Architectural Bodies => Structural + Dataflow + Behavioral. A structural body is a schematic view of subcomponents and interconnections. A dataflow body consists of the data transforms being performed in terms of concurrently executing RTL statements. The need is to express "what" is done. Behavioral descriptions specify data transforms in terms of algorithms. The are delineated by process statements. The keyword memoried is a reserved word used to allow synchronous operation within a block. This is a guard condition. Regular descriptions may contain generate statements. This is considered as taking a subcomponent described by a process statement and replicating it the number of times specified in the generate statement. Configuration specifications allow a designer to use already existing design entities. The design entity may be instantiated and modified as necessary. Under additional language features, it appears that "typing," "functions and procedures," and "packages" are much like the Ada constructs.

Lowenstein, Al and Greg Winter. "VHDL's Impact on Test," *IEEE Design and Test*, 3: 48-53 (April 1986).

A VHDL description of a unit under test (UUT) can be input into an automatic test specification generator (ATSG) to generate a test specification written in a test description language (TDL). The TDL is then sent through a postprocessor to generate a test program in the TDL. Products to be tested can exist at many different levels (whole system, Ics, Pcbs, etc.). Testing requirements at each level are different. However, each level is interrelated with the next and testing should allow for incorporation into a package. VHDL allows this. The emphasis of the article is on testing in the design/development, production, and maintenance phases of the product life cycle. The article further describes the rationale for different types of testing encountered during the different phases. The article provides tables listing the types of tests to be performed during the different portions of the product life cycle and the types of data required to perform the tests. The author then gives examples of different VHDL instances that provide the necessary items to perform some of the tests. Basically, it appears that VHDL covers the electrical, physical, and environmental factors necessary for constructing tests in the different phases of the product life cycle.

Nash, J.D. and L.F. Saunders. "VHDL - Critique," *IEEE Design and Test*, 3: 54-65 (April 1986).

There seems to be some difficulty in determining the best point for a boundary between technology independent and technology dependent information in the package description for a design entity. Therefore, the problem arises in requiring all fully documented designs to rely on the standard package.

Incorporation of a WAIT command within the syntax has been requested as an enhancement. The WAIT would allow for the suspension of sequentially executed statements for a specified period of time before continuing. A problem can arise in optimizing the VHDL readability over function resolution in memoried and nonmemoried signals. VHDL does not posses the ability to use dynamic data like linked lists in Pascal. This leads to misuse of arrays and matrices within VHDL. VHDL does not posses facilities for data transfer into or out of the design entity. The ATOMIC statement is compared to the PRAGMA statement in Ada. Though both provide information to the simulator in the former and compiler in the latter, the ATOMIC statement cannot be removed as the PRAGMA statement since it is embedded in the actual hardware description. VHDL does not allow overloading of operators. Some users expressed a desire for graphics.

Shahdad, M. and others. "VHSIC Hardware Description Language," *Computer*, 18: 94-103 (February 1985).

VHDL was required to be constructed using Ada constructs whenever possible. Three apparent goals of VHDL were support of design, documentation, and efficient simulation from the digital system level to the gate level. The major goal of VHDL is technology insertion of the latest technology into already developed systems. After this brief introduction, the paper begins discussion of the design entity. The design entity consists of the interface and of several bodies.

APPENDIX B: Component Listing of 31-Bit Register Macrocell

The following is the resulting component representation for the 31-Bit register macrocell after extraction. The format of several of the component descriptions is provided.

dff(A,B,C,D,1). := describes a static D flip-flop where A is the enabling clock pulse high, B is the inverted clock pulse, C is the input, D is the inverted output, and 1 describes the D flip-flop as a static CMOS design.

buffer(A,B,1). := describes a buffer where A is the input and B is the output. The 1 indicates a CMOS static buffer consisting of two inverters.

inv(A,B,1). := describes an inverter where A is the input and B is the output. The 1 indicates a CMOS static design.

tgate(A,B,C,D,1). := describes a transmission gate where A is the p-type transistor gate, B is the n-type transistor gate, C is the input/output, and D is the output/input.

```
dff(nPQ2bar,nPQ2,n1903,n573,1).
dff(nPQ2bar,nPQ2,n1812,n572,1).
dff(nPQ2bar,nPQ2,n1811,n571,1).
dff(nPQ2bar,nPQ2,n1810,n570,1).
dff(nPQ2bar,nPQ2,n1809,n569,1).
dff(nPQ2bar,nPQ2,n1808,n568,1).
dff(nPQ2bar,nPQ2,n1807,n567,1).
dff(nPQ2bar,nPQ2,n1806,n566,1).
dff(nPQ2bar,nPQ2,n1805,n565,1).
dff(nPQ2bar,nPQ2,n1804,n564,1).
dff(nPQ2bar,nPQ2,n1803,n563,1).
dff(nPQ2bar,nPQ2,n1802,n562,1).
dff(nPQ2bar,nPQ2,n1801,n561,1).
dff(nPQ2bar,nPQ2,n1800,n560,1).
dff(nPQ2bar,nPQ2,n1799,n559,1).
dff(nPQ2bar,nPQ2,n1798,n558,1).
dff(nPQ2bar,nPQ2,n1797,n557,1).
dff(nPQ2bar,nPQ2,n1796,n556,1).
dff(nPQ2bar,nPQ2,n1795,n555,1).
dff(nPQ2bar,nPQ2,n1794,n554,1).
dff(nPQ2bar,nPQ2,n1793,n553,1).
dff(nPQ2bar,nPQ2,n1792,n552,1).
dff(nPQ2bar,nPQ2,n1791,n551,1).
dff(nPQ2bar,nPQ2,n1790,n550,1).
dff(nPQ2bar,nPQ2,n1789,n549,1).
dff(nPQ2bar,nPQ2,n1788,n548,1).
dff(nPQ2bar,nPQ2,n1787,n547,1).
dff(nPQ2bar,nPQ2,n1786,n546,1).
dff(nPQ2bar,nPQ2,n1785,n545,1).
dff(nPQ2bar,nPQ2,n1784,n544,1).
dff(nPQ2bar,nPQ2,n1783,n543,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN30,n1903,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN29,n1812,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN28,n1811,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN27,n1810,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN26,n1809,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN25,n1808,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN24,n1807,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN23,n1806,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN22,n1805,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN21,n1804,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN20,n1803,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN19,n1802,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN18,n1801,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN17,n1800,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN16,n1799,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN15,n1798,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN14,n1797,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN13,n1796,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN12,n1795,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN11,n1794,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN10,n1793,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN9,n1792,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN8,n1791,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN7,n1790,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN6,n1789,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN5,n1788,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN4,n1787,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN3,n1786,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN2,n1785,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN1,n1784,1).
dff(nLD_PQ1_bar,nLD_PQ1,nIN0,n1783,1).
buffer(n573,nOZ30,1).
buffer(n572,nOZ0,1).
buffer(n571,nOZ1,1).
buffer(n570,nOZ2,1).
buffer(n569,nOZ3,1).
buffer(n568,nOZ4,1).
buffer(n567,nOZ5,1).
buffer(n566,nOZ6,1).
buffer(n565,nOZ7,1).
buffer(n564,nOZ8,1).
buffer(n563,nOZ9,1).
buffer(n562,nOZ10,1).
buffer(n561,nOZ11,1).
buffer(n560,nOZ12,1).
buffer(n559,nOZ13,1).
buffer(n558,nOZ14,1).
buffer(n557,nOZ15,1).
buffer(n556,nOZ16,1).
buffer(n555,nOZ17,1).
buffer(n554,nOZ18,1).
buffer(n553,nOZ19,1).
buffer(n552,nOZ20,1).
buffer(n551,nOZ21,1).
buffer(n550,nOZ22,1).
buffer(n549,nOZ23,1).
buffer(n548,nOZ24,1).
buffer(n547,nOZ25,1).
buffer(n546,nOZ26,1).
buffer(n545,nOZ27,1).
buffer(n544,nOZ28,1).
buffer(n543,nOZ29,1).
inv(nGO,nsbar0,1).
inv(nGO,nsbar1,1).
inv(nGO,nsbar2,1).
inv(nGO,nsbar3,1).
inv(nGO,nsbar4,1).
inv(nGO,nsbar5,1).
inv(nGO,nsbar6,1).
inv(nGO,nsbar7,1).
inv(nGO,nsbar8,1).
inv(nGO,nsbar9,1).
inv(nGO,nsbar10,1).
inv(nGO,nsbar11,1).
inv(nGO,nsbar12,1).
inv(nGO,nsbar13,1).
inv(nGO,nsbar14,1).
inv(nGO,nsbar15,1).
inv(nGO,nsbar16,1).
inv(nGO,nsbar17,1).
inv(nGO,nsbar18,1).
inv(nGO,nsbar19,1).
inv(nGO,nsbar20,1).
inv(nGO,nsbar21,1).
inv(nGO,nsbar22,1).
inv(nGO,nsbar23,1).
inv(nGO,nsbar24,1).
inv(nGO,nsbar25,1).
inv(nGO,nsbar26,1).
inv(nGO,nsbar27,1).
inv(nGO,nsbar28,1).
inv(nGO,nsbar29,1).
inv(nGO,nsbar30,1).
tgate(nsbar0,nGO,gnd,nIN0,1).
tgate(nsbar1,nGO,gnd,nIN1,1).
tgate(nsbar2,nGO,gnd,nIN2,1).
tgate(nsbar3,nGO,gnd,nIN3,1).
tgate(nsbar4,nGO,gnd,nIN4,1).
tgate(nsbar5,nGO,gnd,nIN5,1).
tgate(nsbar6,nGO,gnd,nIN6,1).
tgate(nsbar7,nGO,gnd,nIN7,1).
tgate(nsbar8,nGO,gnd,nIN8,1).
tgate(nsbar9,nGO,gnd,nIN9,1).
tgate(nsbar10,nGO,gnd,nIN10,1).
tgate(nsbar11,nGO,gnd,nIN11,1).
tgate(nsbar12,nGO,gnd,nIN12,1).
tgate(nsbar13,nGO,gnd,nIN13,1).
tgate(nsbar14,nGO,gnd,nIN14,1).
tgate(nsbar15,nGO,gnd,nIN15,1).
tgate(nsbar16,nGO,gnd,nIN16,1).
tgate(nsbar17,nGO,gnd,nIN17,1).
tgate(nsbar18,nGO,gnd,nIN18,1).
tgate(nsbar19,nGO,gnd,nIN19,1).
tgate(nsbar20,nGO,gnd,nIN20,1).
tgate(nsbar21,nGO,gnd,nIN21,1).
tgate(nsbar22,nGO,gnd,nIN22,1).
tgate(nsbar23,nGO,gnd,nIN23,1).
tgate(nsbar24,nGO,gnd,nIN24,1).
tgate(nsbar25,nGO,gnd,nIN25,1).
tgate(nsbar26,nGO,gnd,nIN26,1).
tgate(nsbar27,nGO,gnd,nIN27,1).
tgate(nsbar28,nGO,gnd,nIN28,1).
tgate(nsbar29,nGO,gnd,nIN29,1).
tgate(nsbar30,nGO,gnd,nIN30,1).
tgate(nGO,nsbar0,nIN0,nIZY29,1).
tgate(nGO,nsbar1,nIN1,nIZY28,1).
tgate(nGO,nsbar2,nIN2,nIZY27,1).
tgate(nGO,nsbar3,nIN3,nIZY26,1).
tgate(nGO,nsbar4,nIN4,nIZY25,1).
tgate(nGO,nsbar5,nIN5,nIZY24,1).
tgate(nGO,nsbar6,nIN6,nIZY23,1).
tgate(nGO,nsbar7,nIN7,nIZY22,1).
tgate(nGO,nsbar8,nIN8,nIZY21,1).
tgate(nGO,nsbar9,nIN9,nIZY20,1).
tgate(nGO,nsbar10,nIN10,nIZY19,1).
tgate(nGO,nsbar11,nIN11,nIZY18,1).
tgate(nGO,nsbar12,nIN12,nIZY17,1).
tgate(nGO,nsbar13,nIN13,nIZY16,1).
tgate(nGO,nsbar14,nIN14,nIZY15,1).
tgate(nGO,nsbar15,nIN15,nIZY14,1).
tgate(nGO,nsbar16,nIN16,nIZY13,1).
tgate(nGO,nsbar17,nIN17,nIZY12,1).
tgate(nGO,nsbar18,nIN18,nIZY11,1).
tgate(nGO,nsbar19,nIN19,nIZY10,1).
tgate(nGO,nsbar20,nIN20,nIZY9,1).
tgate(nGO,nsbar21,nIN21,nIZY8,1).
tgate(nGO,nsbar22,nIN22,nIZY7,1).
tgate(nGO,nsbar23,nIN23,nIZY6,1).
tgate(nGO,nsbar24,nIN24,nIZY5,1).
tgate(nGO,nsbar25,nIN25,nIZY4,1).
tgate(nGO,nsbar26,nIN26,nIZY3,1).
tgate(nGO,nsbar27,nIN27,nIZY2,1).
tgate(nGO,nsbar28,nIN28,nIZY1,1).
tgate(nGO,nsbar29,nIN29,nIZY0,1).
tgate(nGO,nsbar30,nIN30,nIZY30,1).
```

APPENDIX C: Code

The following is the code for the sim2pro translation routine.

```
/*******************************************************************
*
*       Date: 16 August 1988
*       Version: 1.0
*
*       Title: sim2pro Translation Routine
*       Filename: sim2pro.c
*       Author: CPT Michael Dukes
*       Project: STOVE_P
*       Operating System: Unix V4.2, Unix V4.3, MS-DOS V3.2
*       Language: C
*       Description:
*               This routine takes a transistor netlist from a sim file
*               generated by mexscp and generates a Prolog formatted
*               description for the same file. The fields used from the
*               sim file are the transistor type, gate, drain, and source.
*               The Prolog output may then be used by any Prolog tool.
*
*       Passed Variables: None
*       Returns: None
*       Global Variables Used: None
*       Global Variables Changed: None
*       Files Read: temp.sim
*       Files Written: good.pro
*       Hardware Input: None
*       Hardware Output: None
*       Modules Called: None
*       Calling Modules: None
*       History: None
*       Special Instructions: This routine should be executed after upper.
*
*******************************************************************/

#include <stdio.h>

#define max_buf 128

char buffer[max_buf];
char tempbuf[max_buf];
```

```
int time_count,
    iteration,
    count,count2;

FILE *fd,*od;

main()
{
fd=fopen("temp.sim","r");
od=fopen("good.pro","w");
for(count=0;count<max_buf;count++)
    {
    tempbuf[count]=0;
    }
fgets(buffer,max_buf,fd);
while(fgets(buffer,max_buf,fd) != NULL)
    {
    if(buffer[0]=='e')
        {
        count=3;
        count2=2;
        iteration=0;
     tempbuf[0]='n';
     tempbuf[1]='(';
        tempbuf[2]='n';
        while((buffer[count]!=0)&(iteration!=3))
            {
            if ((buffer[count2]=='V')&(buffer[count2+1]=='d')&
              (buffer[count2+2]=='d'))
                {
                --count;
                tempbuf[count++]='v';
                tempbuf[count++]='d';
                tempbuf[count]='d';
                count2=count2+2;
                }
            else if
((buffer[count2]=='G')&(buffer[count2+1]=='N')&
                   (buffer[count2+2]=='D'))
                {
                --count;
                tempbuf[count++]='g';
                tempbuf[count++]='n';
                tempbuf[count]='d';
                count2=count2+2;
                }
            else if(buffer[count2]==' ')
                {
                tempbuf[count++]=',';
                tempbuf[count]='n';
                iteration++;
                }
            else if(buffer[count2]=='#')
                {
                --count;
                }
            else
                {
                tempbuf[count]=buffer[count2];
                }
            count++;
            count2++;
            }
            count=count-2;
         tempbuf[count++]=')';
            tempbuf[count++]='.';
            tempbuf[count++]=10;
            tempbuf[count]=0;
        for(count=0;count<max_buf;count++)
            {
            buffer[count]=tempbuf[count];
            }
        fprintf(od,"%s",buffer);
        for(count=0;count<max_buf;count++)
            {
            tempbuf[count]=0;
            }
        }
    else if(buffer[0]=='p')
        {
        count=3;
        count2=2;
```

```
   iteration=0;
tempbuf[0]='p';
tempbuf[1]='(';
   tempbuf[2]='n';
   while((buffer[count]!=0)&(iteration!=3))
       {
       if ((buffer[count2]=='V')&(buffer[count2+1]=='d')&
         (buffer[count2+2]=='d'))
           {
           --count;
           tempbuf[count++]='v';
           tempbuf[count++]='d';
           tempbuf[count]='d';
           count2=count2+2;
           }
       else if((buffer[count2]=='G')&(buffer[count2+1]=='N')&
             (buffer[count2+2]=='D'))
           {
           --count;
           tempbuf[count++]='g';
           tempbuf[count++]='n';
           tempbuf[count]='d';
           count2=count2+2;
           }
       else if(buffer[count2]==' ')
           {
           tempbuf[count++]=',';
           tempbuf[count]='n';
           iteration++;
           }
       else if(buffer[count2]=='#')
           {
           --count;
           }
       else
           {
           tempbuf[count]=buffer[count2];
           }
       count++;
       count2++;
```

```
                }
                count=count-2;
             tempbuf[count++]=')';
                tempbuf[count++]='.';
                tempbuf[count++]=10;
                tempbuf[count]=0;
            for(count=0;count<max_buf;count++)
                {
                buffer[count]=tempbuf[count];
                }
            fprintf(od,"%s",buffer);
            for(count=0;count<max_buf;count++)
                {
                tempbuf[count]=0;
                }
            }
        }
    fclose(fd);
    fclose(od);
    }
```

APPENDIX D: Package Body for Some Common MOS Functions

```
----------------------------------------------------------------
----------------------------------------------------------------
--
--      Date: 1 September 1988
--      Version: 1.0
--
--      Title: Package for Some Standard MOS Logic Functions
--      Filename: mos.vhd
--      Author: CPT Michael Dukes
--      Project: STOVE_P
--      Operating System: VMS V 4.5
--      Language: VHDL
--      Contents:
--              mos_node_resolution
--              snand (A,B : MOS_node_record)
--              snor  (A,B : MOS_node_record)
--              snot  (A : MOS_node_record)
--              sxnor (A,B : MOS_node_record)
--              pnand (A,B : MOS_node_record)
--              nnand (A,B : MOS_node_record)
--              pnor  (A,B : MOS_node_record)
--              pnot  (A : MOS_node_record)
--              tmux  (A,B,S,Sbar : MOS_node_record)
--              dff   (A, PHI, PHI_bar, OUTput : MOS_node_record)
--              binary_to_multi (A : bit)
--              multi_to_binary (Sig : bit; A : MOS_node_record)
--
--      Description:
--              This package contains a collection of functions for
--              performing bus resolution, computing logical output
--              for several types of components, and converting
--              between two-valued and ten-valued logic systems.
--
--      History:
--
----------------------------------------------------------------
----------------------------------------------------------------
```

```
package body MOS_logic_package is

---------- ----------------------------------------------------------
---------------------------------------------------------------------
--
--   Bus resolution function used for mos_node_record
--
---------------------------------------------------------------------
---------------------------------------------------------------------

    function mos_node_resolution
        (input : mos_node_resolution_array)
        return mos_node_record is

        variable output, temp : mos_node_record;

        begin
        output.L.S := 'B';
        output.L.V := 'B';
        for i in input'range loop
            temp := input(i);
            If (temp.L.S > output.L.S) then
                output := temp;
            elsif ((temp.L.S = output.L.S) and
                (temp.L.V /= output.L.V)) then
                output.L.V := 'X';
                end if;
            end loop;
        return(output);
        end mos_node_resolution;
```

```
----------------------------------------------------------------
----------------------------------------------------------------
--
--  Static CMOS functions used with multiple-valued logic
--
----------------------------------------------------------------
----------------------------------------------------------------

    function snand      (A,B : MOS_node_record)
                        return MOS_node_record is

        variable temp : MOS_node_record;

        begin

--          temp.C := 0.0;
        If(A.L.V = '1') and (B.L.V = '1') then
            temp.L.S := 'D';
            temp.L.V := '0';
        elsif (B.L.V = '0') or (A.L.V = '0') then
            temp.L.S := 'D';
            temp.L.V := '1';
        else
            temp.L.S := 'W';
            temp.L.V := 'X';
            end if;
        return(temp);

        end snand;

    function snor       (A,B : MOS_node_record)
                        return MOS_node_record is

        variable temp : MOS_node_record;

        begin

--          temp.C := 0.0;
        If(A.L.V = '0') and (B.L.V = '0') then
            temp.L.S := 'D';
            temp.L.V := '1';
        elsif (B.L.V = '1') or (A.L.V = '1') then
            temp.L.S := 'D';
            temp.L.V := '0';
        else
            temp.L.S := 'W';
            temp.L.V := 'X';
            end if;
        return(temp);

        end snor;

    function snot         (A : MOS_node_record)
                          return MOS_node_record is

        variable temp : MOS_node_record;

        begin

--          temp.C := 0.0;
        If (A.L.V = '0') then
            temp.L.S := 'D';
            temp.L.V := '1';
        elsif (A.L.V = '1') then
            temp.L.S := 'D';
            temp.L.V := '0';
        else
            temp.L.S := 'W';
            temp.L.V := 'X';
            end if;
        return(temp);

        end snot;

    function sxnor        (A,B : MOS_node_record)
                          return MOS_node_record is

        variable temp : MOS_node_record;

        begin

--          temp.C := 0.0;
        If ((A.L.V = '0') and (B.L.V = '0')) or
           ((A.L.V = '1') and (B.L.V = '1')) then
            temp.L.S := 'D';
            temp.L.V := '1';
        elsif ((A.L.V = '1') and (B.L.V = '0')) or
              ((A.L.V = '0') and (B.L.V = '1')) then
    temp.L.S := 'D';
    temp.L.V := '0';
else
    temp.L.S := 'W';
    temp.L.V := 'X';
    end if;
return(temp);

end sxnor;
```

```
-----------------------------------------------------------------------
-----------------------------------------------------------------------
--
--   pnMOS functions used with multiple-valued logic
--
-----------------------------------------------------------------------
-----------------------------------------------------------------------

   function pnand       (A,B : MOS_node_record)
                        return MOS_node_record is

       variable temp : MOS_node_record;

       begin

--         temp.C := 0.0;
       If (A.L.V = '1') and (B.L.V = '1') then
           temp.L.S := 'D';
           temp.L.V := '0';
       elsif (A.L.V = '0') or (B.L.V = '0') then
           temp.L.S := 'W';
           temp.L.V := '1';
       else
           temp.L.S := 'W';
           temp.L.V := 'X';
           end if;
       return(temp);

       end pnand;

   function nnand       (A,B : MOS_node_record)
                        return MOS_node_record is

       variable temp : MOS_node_record;

       begin

--         temp.C := 0.0;
       If (A.L.V = '1') and (B.L.V = '1') then
           temp.L.S := 'W';
           temp.L.V := '0';
       elsif (A.L.V = '0') or (B.L.V = '0') then
           temp.L.S := 'D';
           temp.L.V := '1';
```

```
        else
            temp.L.S := 'W';
            temp.L.V := 'X';
            end if;
        return(temp);

        end nnand;

    function pnor       (A,B : MOS_node_record)
                        return MOS_node_record is

        variable temp : MOS_node_record;

        begin

--          temp.C := 0.0;
        If (A.L.V = '1') or (B.L.V = '1') then
            temp.L.S := 'D';
            temp.L.V := '0';
        elsif (A.L.V = '0') and (B.L.V = '0') then
            temp.L.S := 'W';
            temp.L.V := '1';
        else
            temp.L.S := 'W';
            temp.L.V := 'X';
            end if;
        return(temp);

        end pnor;

    function pnot       (A : MOS_node_record)
                        return MOS_node_record is

        variable temp : MOS_node_record;

        begin

--          temp.C := 0.0;
        If (A.L.V = '1') then
            temp.L.S := 'D';
            temp.L.V := '0';
        elsif (A.L.V = '0') then
            temp.L.S := 'W';
            temp.L.V := '1';
```

```
else
    temp.L.S := 'W';
    temp.L.V := 'X';
    end if;
return(temp);

end pnot;
```

```
-------------------------------------------------------------
-------------------------------------------------------------
--
--   transmission-gate functions used with multiple-valued
--   logic
--
-------------------------------------------------------------
-------------------------------------------------------------

   function tmux        (A,B,S,Sbar : MOS_node_record)
                       return MOS_node_record is

      variable temp : MOS_node_record;

      begin

--       temp.C := 0.0;
      If ((S.L.V = '1') and (Sbar.L.V = '0')) then
         If not(A.L.S = 'B') then
            temp := A;
            end if;
      elsif ((S.L.V = '0') and (Sbar.L.V = '1')) then
         If not(B.L.S = 'B') then
            temp := B;
            end if;
      elsif (A.L.S = B.L.S) then
         If (A.L.V = B.L.V) then
            temp.L := A.L;
         else
            temp.L.V := 'X';
            temp.L.S := A.L.S;
         end if;
         If (temp.L.S = 'D') then
            if (S.L.V = '0') and (temp.L.V = '0') then
               temp.L.S := 'W';
            elsif (S.L.V = '1') and (temp.L.V = '1') then
               temp.L.S := 'W';
            elsif (S.L.V = 'X') then
               temp.L.S := 'W';
            end if;
         end if;
      elsif (A.L.S > B.L.S) then
         temp.L := A.L;
```

```
    If (temp.L.S = 'D') then
        If (S.L.V = '0') and (temp.L.V = '0') then
            temp.L.S := 'W';
        elsif (S.L.V = '1') and (temp.L.V = '1') then
            temp.L.S := 'W';
        elsif (S.L.V = 'X') then
            temp.L.S := 'W';
        end if;
    end if;
else
    temp.L := B.L;
    If (temp.L.S = 'D') then
        If (S.L.V = '0') and (temp.L.V = '0') then
            temp.L.S := 'W';
        elsif (S.L.V = '1') and (temp.L.V = '1') then
            temp.L.S := 'W';
        elsif (S.L.V = 'X') then
            temp.L.S := 'W';
        end if;
    end if;
end if;

return(temp);

end tmux;
```

```
----------------------------------------------------------------
----------------------------------------------------------------
--
--   precharged functions used with multiple-valued logic
--   (to be implemented)
----------------------------------------------------------------
----------------------------------------------------------------


----------------------------------------------------------------
----------------------------------------------------------------
--
--   Other logic functions used with multiple-valued logic
--
----------------------------------------------------------------
----------------------------------------------------------------

   function dff (A, PHI, PHI_bar, OUTput : MOS_node_record)
                        return MOS_node_record is

      variable temp : MOS_node_record;

      begin

--       temp.C := 0.0;
      If ((PHI.L.V = '1' and PHI_bar.L.V = '0')) then
         If (A.L.V = '0') then
            temp.L.S := 'D';
            temp.L.V := '1';
         elsif (A.L.V = '1') then
            temp.L.S := 'D';
            temp.L.V := '0';
         else
            temp.L.S := 'W';
            temp.L.V := 'X';
            end if;
      else
         temp.L := OUTput.L;
         end if;
      return(temp);

      end dff;
```

```
--------------------------------------------------------------------
--------------------------------------------------------------------
--
--   Logic translation for binary to multi and multi to binary
--
--------------------------------------------------------------------
--------------------------------------------------------------------

   function binary_to_multi (A : bit)
                    return MOS_node_record is

      variable temp : MOS_node_record;

      begin

      temp.L.S := 'D';
      If (A = '1') then
         temp.L.V := '1';
      else
         temp.L.V := '0';
         end if;
      return(temp);

      end binary_to_multi;

   function multi_to_binary (Sig : bit;
                         A : MOS_node_record)
                    return bit is

      variable temp : bit;

      begin

      If (A.L.V = '1') then
         temp := '1';
      elsif (A.L.V = '0') then
         temp := '0';
      else
         temp := Sig;
         end if;
      return(temp);
      end multi_to_binary;

       end MOS_logic_package;
```

APPENDIX E: Extracted VHDL Description of the Clock Generator

The following is a listing of the VHDL code automatically generated by STOVE_P for a clock generator.

```
use work.mos_logic_package.all;
entity INV is
    generic(constant tPLH:TIME:=0 ns;
            constant tPHL:TIME:=0 ns);
          port (signal A:in mos_node;
                signal B:out mos_node);
end;
architecture INV of INV is
        begin
            B <= snot(A) after 1ns;
end;
use work.mos_logic_package.all;
entity NAND_GATE is
    generic(constant tPLH:TIME:=0 ns;
            constant tPHL:TIME:=0 ns);
         port  (signal A:in mos_node;
                signal B:in mos_node;
                signal C:out mos_node);
end;
architecture NAND_GATE of NAND_GATE is
        begin
            C <= snand(A,B) after 2 ns;
end;
use work.mos_logic_package.all;
entity NOR_GATE is
    generic(constant tPLH:TIME:=0 ns;
            constant tPHL:TIME:=0 ns);
         port  (signal A:in mos_node;
                signal B:in mos_node;
                signal C:out mos_node);
end;
architecture NOR_GATE of NOR_GATE is
        begin
            C <= snor(A,B) after 2 ns;
end;
entity test is
end test;
use work.mos_logic_package.all;
use std.simulator_standard.all;
architecture testor of test is
        signal n444 : mos_node;
        signal n177 : mos_node;
        signal n498 : mos_node;
        signal nIZ_go : mos_node;
        signal n450 : mos_node;
        signal n453 : mos_node;
        signal n424 : mos_node;
        signal n443 : mos_node;
        signal n449 : mos_node;
        signal n533 : mos_node;
        signal n584 : mos_node;
        signal n447 : mos_node;
        signal n233 : mos_node;
        signal n277 : mos_node;
        signal n329 : mos_node;
        signal nOZ_pq2 : mos_node;
        signal n53 : mos_node;
        signal nOZ_pq1 : mos_node;
        signal n128 : mos_node;
        signal gnd : mos_node;
        signal vdd : mos_node;
            component INV
                generic ( constant tPLH: TIME;
                    constant tPHL: TIME);
                port ( signal A: in mos_node;
                signal B: out mos_node);
            end component;
        for all : inv use entity work.inv( inv );
```

```
                component NAND_GATE
                        generic ( constant tPLH: TIME;
                            constant tPHL: TIME);
                        port ( signal A: in mos_node;
                        signal B: in mos_node;
                        signal C: out mos_node);
                    end component;
            for all : nand_gate use
                entity work.nand_gate( nand_gate );
                component NOR_GATE
                        generic ( constant tPLH: TIME;
                            constant tPHL: TIME);
                        port ( signal A: in mos_node;
                        signal B: in mos_node;
                        signal C: out mos_node);
                    end component;
            for all : nOr_gate use
                entity work.nor_gate( NOr_gate );
begin
process
    variable high_volt  : mos_node_record;
    variable low_volt   : mos_node_record;
    begin
    set_maximums(10000,100);
    tracing_on;
    high_volt.L.S := 'D';
    high_volt.L.V := '1';
    low_volt.L.S  := 'D';
    low_volt.L.V  := '0';
    vdd <= high_volt;
    gnd <= low_volt;
    end process;
            INV1:INV
                generic map( tPLH => 0 ns,
                    tPHL => 0 ns)
                port map( A =>n128,
                    B =>nOZ_pq1);
            INV2:INV
                generic map( tPLH => 0 ns,
                    tPHL => 0 ns)
    port map( A =>n53,
        B =>nOZ_pq2);
INV3:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n329,
        B =>n277);
INV4:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n233,
        B =>n53);
INV5:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n447,
        B =>n584);
INV6:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n533,
        B =>n128);
INV7:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n449,
        B =>n443);
INV8:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n277,
        B =>n233);
INV9:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n584,
        B =>n533);
INV10:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n424,
        B =>n453);
INV11:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n450,
        B =>n424);
INV12:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>nIZ_go,
        B =>n498);
INV13:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n177,
        B =>n444);
INV14:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n443,
        B =>n177);
INV15:INV
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A =>n444,
        B =>n450);
NAND_GATE1: NAND_GATE
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A => n498,
        B => n450,
        C => n449);
NOR_GATE1: NOR_GATE
    generic map( tPLH => 0 ns,
        tPHL => 0 ns)
    port map( A => n329,
        B => n424,
        C => n447);
NOR_GATE2: NOR_GATE
```

```
                    generic map( tPLH => 0 ns,
                        tPHL => 0 ns)
                    port map( A => n447,
                        B => n453,
                        C => n329);
        end testor;
```

BIBLIOGRAPHY

Aylor, J.H. "VHDL - Feature Description and Analysis," *IEEE Design and Test*, 3: 17-27 (April 1986).

Barna, Arpad. *VHSIC Technologies and Tradeoffs*. New York: John Wiley & Sons, 1981.

Barr, Avron and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume 1*. Los Altos: William Kaufmann, Inc., 1981.

_____. *The Handbook of Artificial Intelligence, Volume 2*. Los Altos: William Kaufmann, Inc., 1981.

Bloch, Norman J. *Abstract Algebra with Applications*. Englewood Cliffs: Prentice-Hall, Inc. 1987.

Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. Wokingham: Addison-Wesley Publishing Company, 1987.

Breuer, Melvin A. "A Note on Three-Valued Logic Simulation," *IEEE Transactions on Computers*, C-21 399-402 (April 1972).

Bryant, Randal E. "MOSSIM: A Switch-Level Simulator for MOS LSI," *Proceedings of the Design Automation Conference*: 786-790 (June 1981).

_____. "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Transactions on Computers*, C-33 160-177 (February 1984).

California, University of Berkeley. Berkeley Distribution of VLSI Design Tools. Computer Science Division, EECS Department, University of California at Berkeley, 1986.

Cha, Charles W., William E. Donath, and Fusun Ozguner. "9-V Algorithm for Test Pattern Generation of Combinational Digital Circuits," *IEEE Transactions on Computers*, C-27 193-200 (March 1978).

Chang, Chin-Liang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Boston: Academic Press, 1973.

Clocksin, W.F. "Logic Programming and Digital Circuit Analysis," *Journal of Logic Programming*, Vol 4. No. 1, 59-82, 1987.

Clocksin, W.F. and C.S. Mellish. *Programming in Prolog*. New York: Springer-Verlag, 1987.

Coelho, David. "Formal Presentations on Desirable Model Features," Integrated Design Automation System (IDAS) Conference, VHDL Model Group. University of Cincinnati. 19 August 1987.

Cohen, Paul R. and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence, Volume 3.* Los Altos: William Kaufmann, Inc., 1981.

d'Abreu, Michael. "Gate-Level Simulation," *IEEE Design and Test*, 2: 63–71 (December 1985).

DeGroat, Major Joseph W., Instructor of Electrical and Computer Engineering. Video Tape Lecture. "VHDL Design Entities." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987.

Dewey, Al and Anthony Gadient. "VHDL Motivation," *IEEE Design and Test*, 3: 12–16 (April 1986).

Gallagher, Captain David M. Rapid Prototyping of Application Specific Processors. MS Thesis, AFIT/GE/ENG/87D–19. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1987.

Gilman, Alfred S. "VHDL - The Designer Environment," *IEEE Design and Test*, 3: 42–47 (April 1986).

Hamachi, Gordon and others. "Magic," 1986 VLSI Tools. 1986.

Hayes, John P. "A Unified Switching Theory with Applications to VLSI Design," *Proceedings of the IEEE*. 1140–1151. 1982.

_____. "A Systematic Approach to Multivalued Digital Simulation," *IEEE International Conference on Computer Design: VLSI in Computers* 177–182. Port Chester, New York, 1984.

_____. "Uncertainty, Energy, and Multiple-Valued Logics," *IEEE Transactions on Computers*, C-35 107–114 (February 1986).

IEEE, Computer Society Standards Committee, "IEEE Standard VHDL Language Reference Manual," *ANSI/IEEE Std 1076–1987*, IEEE Press, New York, 1987.

Intermetrics, Inc. *VHDL User's Manual: Volume I Tutorial*. U.S. Air Force Contract F33615-83-C-1003. Bethesda, Md., 1 August 1985.

Intermetrics, Inc. *VHDL User's Manual: Volume II – User's Reference Guide*. U.S. Air Force Contract F33615-83-C-1003. Bethesda, MD, 30 September 1987a.

Intermetrics, Inc. *VHDL Language Reference Manual.* U.S. Air Force Contract F33615-83-C-1003. Bethesda, MD, 1 January 1987b.

Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* New Jersey: Prentice-Hall, Inc., 1978.

Linderman, Captain Richard, Assistant Professor of Electrical and Computer Engineering. Video Tape Lecture. "Types and Objects in VHDL." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987a.

-----. Video Tape Lecture. "Modeling Circuits in VHDL." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987b.

-----. Video Tape Lecture. "VHDL Process Statements." School of Engineering, Air Force Institute of Technology, Wright- Patterson AFB OH, 1987c.

-----. Video Tape Lecture. "VHDL Examples." School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1987d.

Lipsett, L., E. Marschner, and M. Shahdad. "VHDL - The Language," *IEEE Design and Test*, 3: 29-41 (April 1986).

Lowenstein, Al and Greg Winter. "VHDL's Impact on Test," *IEEE Design and Test*, 3: 48-53 (April 1986).

Lynch, Major William L. VHDL Prototype Simulator. MS Thesis, AFIT/GCS/ENG/86D-15. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

Muth, Peter. "A Nine-Valued Circuit Model for Test Generation," *IEEE Transactions on Computers*, C-25 630-636 (June 1976).

Nash, J.D. and L.F. Saunders. "VHDL - Critique," *IEEE Design and Test*, 3: 54-65 (April 1986).

Obermeier, Fred. "Spice 2 Summary," 1986 VLSI Tools. 1986.

Rubin, Steven M. *Computer Aids for VLSI Design.* Reading: Addison-Wesley Publishing Company, 1987.

Shahdad, M. and others. "VHSIC Hardware Description Language," *Computer*, 18: 94-103 (February 1985).

Stanat, Donald F. and David F. McAllister. *Discrete Mathematics in Computer Science*. Englewood Cliffs: Prentice-Hall, Inc., 1977.

Terman, Chris. Computer Program. "ESIM." 1980.

Terman, Chris. "Esim," 1986 VLSI Tools. 1986.

Ullman, Jeffrey D. *Computational Aspects of VLSI*. Rockville: Computer Science Press, 1984.

Waite, Mitchell, Stephen Prata, and Donald Martin. *C Primer Plus*. Indianapolis: Howard W. Sams and Company, 1987.

Walker, Elbert A. *Introduction to Abstract Algebra*. New York: Random House, 1987.

Watanabe, Joe and others. "Seven Value Logic Simulation for MOS LSI Circuits," *Proceedings of the International Conference on Circuits and Computers*: 941-944 (1980).

Weste, Neil and Kamran Eshraghian. *Principles of CMOS VLSI Design*. Reading: Addison-Wesley Publishing Company, 1985.

Vita

Captain Michael A. Dukes was born [redacted] Following graduation from high school at Springfield, Virginia in 1978, he received an appointment to the US Military Academy at West Point, New York. He graduated from the Military Academy in May 1982, with a degree of Bachelor of Science and a commission in the US Army. After completion of the SignalOfficer Basic Course, Captain Dukes was assigned to Fort Gordon as a Teleprocessing Operations and Computer Automation Officer within the Directorate of Automation and Information Management. Prior to entering the Air Force Institute of Technology, he attended the Signal Officer Advanced Course. Following graduation, Captain Dukes will remain at the Air Force Institute of Technology to pursue the degree of Doctor of Philosophy in VLSI Engineering.

[redacted]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GE/ENG/88S-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering — 6b. OFFICE SYMBOL *(If applicable)* AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS *(City, State, and ZIP Code)* Air Force Institute of Technology (AU) Wright-Patterson AFB, Ohio 45433-6583	7b. ADDRESS *(City, State, and ZIP Code)*
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Air Force Wright Aero Lab — 8b. OFFICE SYMBOL *(If applicable)* AFWAL/AADE-3	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Wright-Patterson AFB, Ohio 45433	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A MULTIPLE-VALUED LOGIC SYSTEM FOR CIRCUIT EXTRACTION TO VHDL 1076-1987 (UNCLASSIFIED)

12. PERSONAL AUTHOR(S)
Michael A. Dukes, B.S., Captain, USA

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
MS Thesis	FROM ______ TO ______	1988 September	166

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	VHDL, Logic Extraction, VLSI, Computer Aided Design
09	01		Multiple-Valued Logic, Prolog Applications

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

Thesis Chairman: Joseph DeGroat, MSEE, Major, USAF
Instructor in Electrical and Computer Engineering

[illegible]
12 Jan 1989

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT.] ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Joseph DeGroat, MSEE, Major, USAF	22b. TELEPHONE *(Include Area Code)* 513-255-6913 — 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. ABSTRACT

Multiple-valued logic is a topic of concern for modeling standards in the VHSIC Hardware Description Language, VHDL 1076-1987. With the various forms of layout styles in MOS devices, there exist different strengths of electrical signals propagated throughout a circuit. Additionally, logic extraction to VHDL of VLSI layout designs may contain leftover transistors that must be modeled correctly in VHDL. A multiple-valued logic system can adequately model signals with different strengths as well as conflicts between signal values. Once a multiple-valued logic system is defined, a logic extraction system may then produce VHDL for hardware component representations down to the transistor level. The goal of this thesis is to present a ten-level multiple-valued logic system and provide a Prolog-based logic extraction tool for generation of VHDL from a transistor netlist. The Prolog-based logic extraction system will also provide groundwork for further research in the area of formal verification with VHDL. Various tools using symbolic representations and multiple-valued logic are essential in a CAD environment where logic extraction from layout to VHDL is incorporated into validation and verification.